
Joe Pritchard
Lenguaje Máquina
para MSX
Introducción y
Conceptos avanzados
SONY
HiT BiT
MSX

LENGUAJE MAQUINA PARA MSX

Introducción y conceptos avanzados

Lenguaje máquina para MSX

Introducción y conceptos avanzados

Joe Pritchard



MICROINFORMATICA

Título de la obra original
MSX MACHINE LANGUAGE FOR THE ABSOLUTE BEGINNER

Traducción de: Alicia Nieto y Juan J. García
Diseño de colección: Antonio Lax
Diseño de cubierta: Narcís Fernández

Primera edición, noviembre 1985
Primera reimpresión, noviembre 1986
Segunda reimpresión, junio 1988

Edición publicada por acuerdo con Melbourne House (Publishers) Ltd., Londres.

Josefa Valcárcel, 27. 28027 Madrid
Depósito legal: M. 22.509-1988
ISBN: 84-7614-058-4
Printed in Spain
Imprime: Anzos, S. A. - Fuenlabrada (Madrid)

Indice

Prólogo

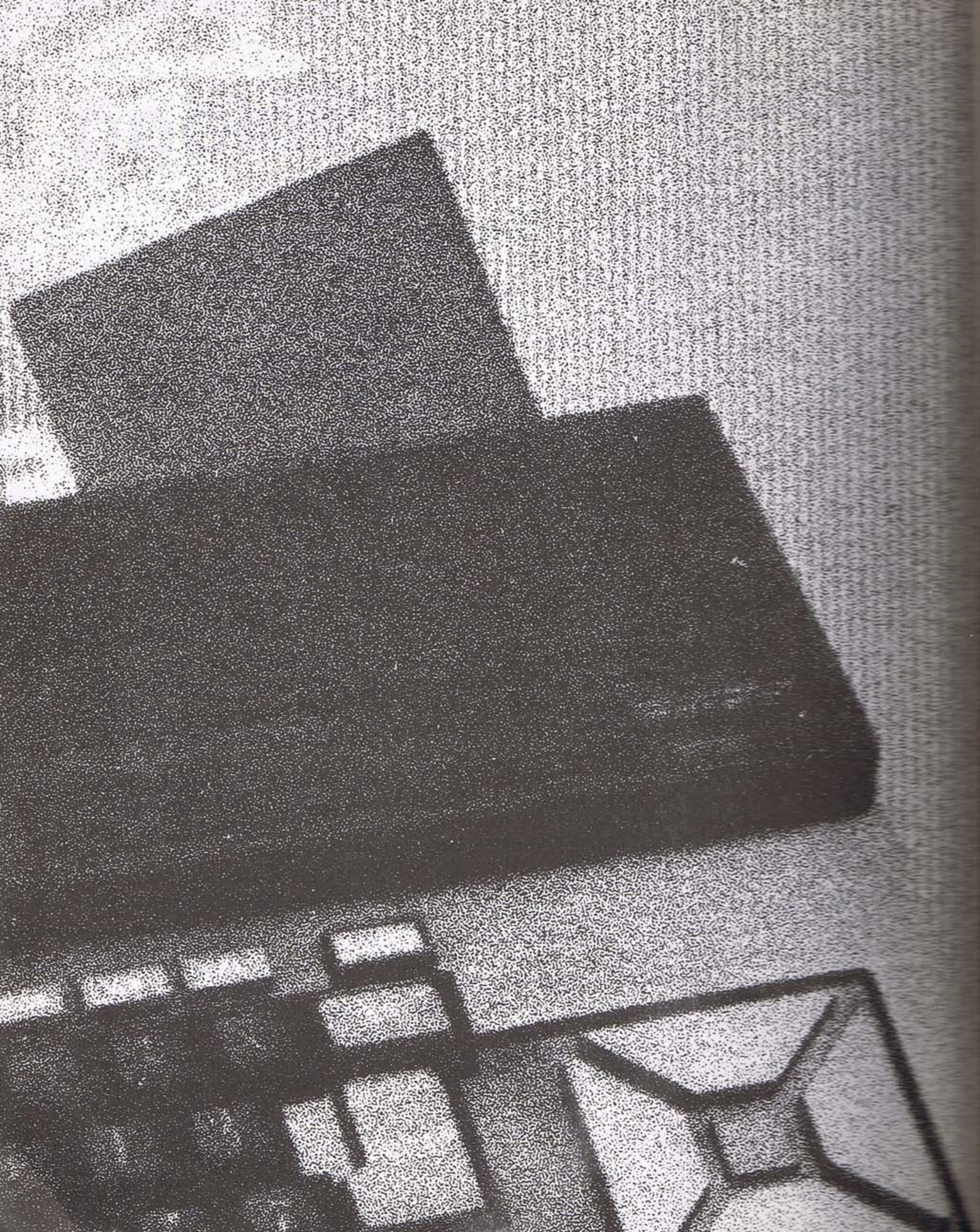
El propósito que persigue este libro es enseñarte a programar en el código máquina de los ordenadores MSX, es decir, programar el microprocesador Z-80. Cuando digo que vas a aprender a programar, me refiero a que, al final de haber estudiado este libro, serás capaz de hacer por ti mismo rutinas que utilicen todo lo que te ofrece tu MSX.

Espero que el libro sea interesante no sólo para los que seáis principiantes en código máquina, para los que está pensado este libro, sino también para todos aquellos que queráis aplicar los conocimientos ya adquiridos sobre el Z-80 para programar en vuestro MSX. Aunque actualmente en el mercado ya hay varios libros sobre este mismo tema, me atrevo a decir que el que tenéis en las manos os será muy útil a todos los que tengáis un ordenador MSX, independientemente de si sabéis algo sobre lenguaje máquina o no.

Para terminar, me gustaría dar las gracias a algunas de las personas que han colaborado en el desarrollo del libro: a mi editor y a todo el personal que con él trabaja, a mi familia y especialmente a mi esposa por su infinito apoyo durante todo el tiempo que he pasado semienclaustrado elaborando este libro como si fuera más un ermitaño que un marido. También se lo dedico a Anna Madill, aunque ella trabaje con ordenadores algo mayores que los MSX, a Elizabeth Madill y a Jason Pritchard, quien demostró su interés por el MSX ayudándome en una parte del libro.

JOE PRITCHARD

Nottingham, octubre 1984.



1

Principios básicos

Este libro fue diseñado para introducir a cualquier programador en MSX BASIC a la lengua, digamos nativa, de su ordenador: el lenguaje máquina o código máquina. Puede que ya sepas algo sobre dicho lenguaje o que sea totalmente nuevo para ti; de cualquier manera, no te preocupes, porque este libro te introducirá paso a paso las ideas y conceptos más fundamentales sobre dicho lenguaje máquina.

Lo primero que vas a hacer, para hacerte una idea de lo que es un lenguaje en código máquina, es ver lo que ocurre cuando utilizas tu ordenador. Si te fijas, lo que haces es teclear varias líneas en lenguaje BASIC, con lo que le ordenas que haga una determinada cosa; sin embargo, y aunque te lo parezca, todavía no estás comunicándote realmente con el cerebro del ordenador, es decir, con la unidad central de proceso (*Central Processor Unit,* CPU en inglés, UCP en español). De hecho, si siempre programases en BASIC, nunca ordenarías nada directamente a la UCP, sino que usarías siempre una especie de intermediario llamado sistema operativo *(Operating System),* del que hablaré más adelante.

La UCP que usa el ordenador MSX se llama Z-80, y en este momento seguramente es la UCP más conocida y extendida de todas las que existen. Hay además otros *chips* electrónicos dentro del MSX, pero se puede decir que la UCP está en el "fondo" de todas las operaciones que ejecuta el ordenador. De hecho, cuando hablamos de programar el MSX en lengüaje máquina, estamos realmente hablando de programar la UCP del ordenador en código máquina Z-80.

¿Qué es código máquina?

El Z-80, por si no has visto todavía ninguno, es un gran *chip* negro con 40 pines (patillas). De estos pines, los más importantes son los ocho con los que la UCP se comunica con el resto del ordenador. Obviamente, la UCP se comunica con el resto del ordenador con señales eléctricas. El Z-80 ha sido diseñado para que actuara dependiendo de las combinaciones eléctricas que va recibiendo en esos 8 pines.

Ya que estamos hablando de señales eléctricas, vamos a ver cómo se representa con números la ausencia o presencia de señal eléctrica: el "1" se utiliza para representar que sí llega señal, es decir, hay presencia de señal, y el "0" denota ausencia de señal. Como hay 8 pines por los que se reciben señales eléctricas, es lógico pensar que habrá 8 señales eléctricas, una por cada pin; por ejemplo, una combinación de señales eléctricas podría ser la siguiente:

01101101

Esta combinación de señales hace que la UCP se comporte de una determinada manera, es decir, dicha combinación ordena a la UCP que ejecute un determinado trabajo; por tanto, dicha combinación decimos que es una instrucción en lenguaje máquina.

Una instrucción en código máquina es la combinación de señales eléctricas del mismo modo que una instrucción en lenguaje BASIC es:

LET A = 0

Básicamente, una instrucción en código máquina es una combinación de señales eléctricas capaces de hacer que la UCP ejecute un determinado trabajo.

Todas las instrucciones que la UCP Z-80 entiende se conocen como "JUEGO DE INSTRUCCIONES" *(instruction set)* del Z-80. Cada UCP tiene un juego de instrucciones diferentes, por lo que un programa escrito en lenguaje máquina Z-80 NO funcionaría adecuadamente en otra UCP.

¿Para qué sirve aprender lenguaje máquina?

Como te habrás dado ya cuenta, la instrucción escrita en código máquina que te he enseñado antes carece de significado para nosotros; por tanto, es lógico que te preguntes: ¿para qué molestarme en aprender código máquina si al fin y al cabo no lo entiendo? Pues bien, hay tres importantes ventajas que el código máquina te ofrece, aventajando al BASIC:

1. Los programas en código máquina son más rápidos.
2. Los programas en lenguaje máquina ocupan menos memoria.
3. No necesitas que tus programas pasen por el sistema operativo del MSX, por lo que actúan directamente sobre la UCP.

Aquí aparece otra vez el sistema operativo MSX; vamos a ver en qué consiste esta parte tan importante para el ordenador.

El sistema operativo

El sistema operativo es un programa en lenguaje máquina cuyo trabajo más importante, en el MSX, consiste en traducir a lenguaje máquina las instrucciones que reciba en BASIC. Una vez que las instrucciones en BASIC estén en código máquina, la UCP ya puede trabajar con ellas y, por tanto, entenderlas. Como ves, la UCP no entiende BASIC, por lo que antes de que pueda empezar a funcionar hay que traducirle todo a lenguaje máquina. Es como cuando alguien nos habla, por ejemplo, en francés y no le entendemos; para poder comunicarnos con él necesitamos o bien un buen diccionario, o bien un intérprete o un traductor que nos vaya traduciendo el francés a español; pues bien, del mismo modo la parte del sistema operativo que realiza ese trabajo de traducción se llama intérprete del BASIC.

Otras partes del sistema operativo indican, por ejemplo, a la UCP cómo controlar la pantalla de la televisión y el teclado.

El hecho de que se tengan que traducir las instrucciones en BASIC antes de poder trabajar con ellas explica por qué es más lento utilizar programas escritos en BASIC que los escritos en lenguaje máquina. El proceso de traducir de un lenguaje a otro lleva tiempo y a menudo las instrucciones en código máquina resultantes no son tan eficientes para el tipo de función que las pensaste. También podemos usar instrucciones en código máquina para llevar a cabo operaciones que el sistema operativo no sería capaz de hacer por no estar diseñado para ello.

Desventajas del código máquina

Algunas desventajas que tiene el código máquina son las siguientes:

1. Los programas en lenguaje máquina son a la vez difíciles de leer y de detectar los errores que tengan.
2. Es muy difícil, acaso imposible, usarlos en otros ordenadores.
3. Tus programas en este lenguaje necesitan de un número muy grande de instrucciones.
4. Los programas que usan aritmética complicada son bastante difíciles de programar en código máquina.

Una vez que ya has visto tanto las ventajas como las desventajas de un lenguaje y de otro estás en posición de elegir por ti mismo si vas a escoger el lenguaje máquina o el BASIC para tus diferentes aplicaciones. Sería erróneo que escribieses una contabilidad en lenguaje máquina, como también lo sería escribir un programa en BASIC si quieres que su velocidad sea muy grande.

Los lenguajes ensambladores

A la mayoría de vosotros el hecho de tener que escribir programas en lenguaje máquina, es decir, como cadenas de unos y ceros, no os atraerá a primera vista por lo complicado de esta notación, así que es lógico que se hayan ideado otras formas de representar estas instrucciones tan complicadas, además de como combinaciones de ceros y unos.

Mientras que para la UCP esas combinaciones de ceros y unos representan varias cosas, para nosotros sólo representan números en código binario; como sabes, cualquier número en binario se puede pasar a su correspondiente en decimal, con lo que ya es más fácil representar dichas instrucciones, pero recuerda que mientras esta nueva forma de representar señales eléctricas es más inteligible para nosotros, la UCP sigue "viendo" dichos números como combinaciones de señales eléctricas que recibe a través de los ocho pines de la UCP.

Por tanto, podemos escribir programas en lenguaje máquina como series de números decimales y así, al menos, son algo más familiares para nosotros y, como ya te dije, más fáciles de representar, aunque todavía no veamos más que una serie de números y no entendamos a qué instrucciones representan para la UCP. A menos que tengamos una lista con los números y sus correspondientes instrucciones, en código máquina difícilmente sabremos lo que quieren decir esas ristras de números para la UCP.

A algunos informáticos de habla inglesa se les ocurrió que sería mucho más útil representar de algún modo, y en inglés, esas señales eléctricas; así fue como nació el lenguaje ensamblador. Cada instrucción en código máquina se puede representar con un término corto llamado MNEMONICO (es decir, fácil de recordar); cada término mnemónico también se conoce como instrucción en ensamblador.

Ya tenemos, pues, tres formas de representar una misma instrucción en lenguaje máquina, que son:

1. Instrucciones en base dos, es decir, en código binario; ejemplo:

   01110110

2. Instrucciones en base diez o decimales; ejemplo:

   118

3. Instrucciones en ensamblador; ejemplo:

   HALT

Si sabes algo de inglés, puede que ya conozcas lo que hace la instrucción en ensamblador: le indica a la UCP que pare lo que esté haciendo, ¡ALTO!, hasta que le diga otra cosa.

El lenguaje ensamblador es, como ya habrás adivinado, ininteligible para la UCP, así que habrá que traducir a instrucciones en lenguaje máquina las que estén

en ensamblador. Puedes hacer la traducción o bien usando las tablas que hay al final del libro o bien con un programa del ordenador que haga este trabajo por ti. Dicho programa se llama ENSAMBLADOR. Se llama ensamblar a mano un programa cuando eres tú quien traduce las instrucciones en ensamblador a código máquina.

Si quieres ver lo que puedes hacer con el código máquina, teclea en tu ordenador el programa en BASIC siguiente. Los números que aparecen en las instrucciones DATA representan instrucciones en lenguaje máquina. Este programa llena toda la pantalla con la letra "A", incluidas aquellas zonas donde tú no llegarías usando la orden PRINT.

```
10 SCREEN 0
20 DEFUSR=58000
30 FOR I=58000 TO 58024
40 READ N:POKE I,N
50 NEXT
60 L=USR(65)
70 END
90 DATA 33,192,3,219,153,62,0,211,153,62,64,211
100 DATA 153,58,248,247,211,152,43,135,180,194
110 DATA 157,226,201
```

Quizá quieras simular algo parecido a lo que hace este programa usando la orden PRINT del BASIC; con ello te darás cuenta de que la rutina en código máquina es mucho más rápida.

No te preocupes si no entiendes todavía el programa anterior; más adelante te lo explicaré con más detalle. Por ahora, considéralo como una muestra de lo que es capaz de hacer el código máquina. Supongo que ya debes estar haciéndote una idea de lo que puedes hacer en código máquina; veamos ahora lo que la UCP del Z-80 es capaz de hacer.

¿Qué hace la UCP?

Se puede decir que la UCP es la responsable de todo lo que ocurra dentro del ordenador; en cuanto lo enciendes, la UCP empieza a ejecutar el sistema operativo, lo que te permite introducir programas e instrucciones en BASIC.

Lo primero que debes tener en cuenta sobre la UCP es que sólo está capacitada para llevar a cabo algunos trabajos muy simples; en segundo lugar, mientras nosotros usamos lápiz y papel para hacer dichas operaciones, el ordenador usa, como los niños, sus dedos para contar. La UCP usa lápiz y papel sólo cuando quiere recordar el resultado de un problema; entonces almacena dicho resultado en cajas o en posiciones dentro de la memoria del ordenador. Por ejemplo, para sumar dos números, digamos un 2 y un 3, la UCP almacena el número 3 en una caja, por ejemplo en la caja número 1, y el número 2 en la caja número 2, y el resultado de la suma en otra, por ejemplo en la caja número 3.

Hay dos cosas que obviamente se deducen de la "cuenta de la vieja" que la UCP usa para ejecutar sus operaciones:

1. La primera es que la UCP sólo trabaja con números enteros; como es lógico, al contar con dedos no puede hacerlo con medio dedo.
2. La segunda es que los números que intervienen en las operaciones pueden ser tan grandes como el número máximo que la UCP pueda contar con sus dedos; es decir, tienen una magnitud limitada.

Para números más grandes, la UCP se sirve no sólo de los dedos de las manos, sino también de los dedos de los pies; pero, mientras que nosotros sólo podemos contar con dos pies y dos manos, la UCP tiene bastantes más a su disposición; además, en vez de tener 5 dedos en cada mano, posee 8 dedos y puede contar hasta el número 255. Puede que pienses que hay algún truco o malabarismo cuando te diga que la UCP puede contar hasta 65535 con los 16 dedos de cada pie.

En el próximo capítulo te revelaré cómo realiza estos cálculos la unidad central de procesos.

La suma antes mencionada se puede escribir en ensamblador y queda como el programa siguiente; date cuenta de que he usado una de las manos con 8 dedos de la UCP.

```
LD      A,3
LD      (CAJA1),A     ; pone el valor 3 en la caja 1
LD      A,4
LD      (CAJA2),A     ; pone el valor 4 en la caja 2
LD      A,(CAJA1)     ; coge el valor 3 y lo pone en la mano A
ADD     (CAJA2)       ; suma al contenido de la mano A lo que hay
                        en la caja 2
LD      (CAJA3),A     ; pone el resultado en la caja 3
```

LD es la abreviatura de *LoaD* (en español, cargar); simplemente estamos cargando la mano A con el valor 3; es decir, cuenta hasta 3 con la mano en la primera instrucción. La segunda instrucción, la que traspasa el contenido de una mano a una caja, es muy importante, ya que introduce un concepto nuevo: el de transferencias. Los paréntesis en esa instrucción indican que queremos trabajar con el contenido de la caja que hay dentro de los paréntesis y no con la caja; en este ejemplo, dentro de la caja 1 se pone la cuenta que realizó la mano A.

Aunque el concepto de caja puede que te recuerde al de variable en BASIC, ten en cuenta que una caja NO es lo mismo que una variable, es simplemente una posición en la memoria del ordenador que se usa como una posición de almacenamiento.

La pila (*stack*)

A pesar de que la UCP tiene 8 manos con 8 dedos cada una y 2 pies con 16 dedos por pie, aún necesita otro tipo de mano como almacén. Aunque normalmente

solemos usar las cajas como almacenes, también pudiera ser que necesitases otro tipo de almacenes; así, apareció como almacén alternativo la pila (*stack,* en inglés). Por ahora hazte a la idea de que una pila es como una barra puntiaguda que la gente ordenada tiene en sus mesas de trabajo para atravesar papeles y se quedan así ''apilados''. Los papeles se empujan (*push,* en inglés), quedando atravesados en la pila, por lo que es obvio pensar que el último papel que se atravesó será el más accesible.

Gracias a la pila, entre otras ventajas, la UCP siempre sabe cuál fue la última información que se añadió en la pila.

La UCP almacena la información de sus manos en la pila siempre que quiere usar dicha mano para otra función; así, cuando quiere volver a recoger esa información, no tendrá más que dar un golpe (*pop,* en inglés) a la última información que dejó, y ésta será devuelta a sus manos.

La UCP puede almacenar la información de tantas manos y pies como desee; cada operación de almacenaje requiere un empujón *(push),* es decir, atravesar el papel en la alcayata; para recuperar la información, también tiene que dar un golpe *(pop)* para cada una de las informaciones que quiera sacar de la pila.

La pila que usa la UCP está patas arriba y cuanta más información añadas en ella, más crecerá hacia abajo dentro de la memoria del ordenador.

¿Qué hace la UCP?

Como ya mencioné al principio del capítulo, la UCP no es capaz de hacer operaciones o trabajos complicados, ya que sus habilidades para contar son muy limitadas, y, debido a que tiene un número también limitado de manos y dedos, los números con los que trabaja la UCP Z-80 son:

Números de 8 dedos entre 0 y 255.
Números de 16 dedos entre 0 y 65535.

En los números representados con 16 dedos no me refiero a números representados con los dedos de los pies, sino con los de las manos; de hecho, se puede conseguir que la UCP use dos de sus manos a la vez para representar un solo número. Un número representado con dos manos es igual al representado por ''unos pies''.

Los tipos de instrucciones que puede ejecutar la UCP son:

1. Contar con una mano.
2. Contar con dos manos.
3. Sumar y restar con una mano.
4. Sumar y restar números de ''dos manos''.
5. Otros cálculos con números de una mano, como, por ejemplo: hacer un número negativo.
6. Conseguir que la UCP salte de un punto a otro dentro de un programa en código máquina.

7. Hacer que la UCP intercambie números en 8 dedos desde y a otros dispositivos del sistema.

Antes de dejar este capítulo sobre principios básicos, paso ya a explicaros algunos conceptos que os serán muy útiles en la programación en código máquina. Dichos conceptos son los de dirección y algunos detalles muy resumidos sobre el *hardware* del MSX.

Direcciones

Seguramente te habrá llamado la atención el término dirección; pues bien, en términos informáticos, una dirección significa más o menos lo mismo que en español; es decir: sitio donde se puede encontrar algo. Pero mientras que en español la dirección de algo se refiere normalmente a una casa o un edificio, en informática se refiere a una posición dentro de la memoria del ordenador donde se encuentra un número; por ejemplo: una instrucción en código máquina.

En los ordenadores MSX las primeras 32000 direcciones diferentes de la memoria contienen números que representan instrucciones en código máquina, que a su vez forman el programa que se ha dado en llamar sistema operativo.

Una posición de memoria sólo puede tener un número de 8 dedos; los números estarán dentro del rango de 0 a 255. Las cajas que comentaba más arriba eran posiciones de memoria y cada una de ellas ocupaba una dirección diferente.

El *hardware* del MSX

Ya hemos visto las principales ideas sobre programación en lenguaje máquina. Antes de pasar a otros temas, como son saber cómo cuenta el ordenador, etc., creo que sería conveniente dar una breve explicación sobre los diferentes componentes electrónicos que forman el sistema MSX. El término *hardware* engloba todos los componentes electrónicos, digamos físicos, que forman parte de un ordenador, y *software* es el nombre inglés con el que se conocen los programas que utilizamos nosotros o que sólo el ordenador usa. Hay algunas personas que para explicar de una forma más fácil la diferencia entre estos dos términos dicen que al *hardware* sí le puedes dar una patada, mientras que al *software* no; pero la verdad es que no estoy muy seguro de que esta explicación me satisfaga.

La unidad central de proceso (UCP)

Hasta aquí hemos visto lo que hace la UCP: ahora veremos su disposición interna. Dentro de la UCP hay 8 manos, llamadas A, B, C, D, E, F, H, L, y

2 pies, llamados IX e IY. Las manos y los pies se representan normalmente como sigue:

A	F
B	C
D	E
H	L
IX	
IY	

Todas las manos, excepto la F, pueden usarse para contar; la mano F tiene una función, digamos, especial: la de indicar, con cada dedo de su mano, si ha ocurrido alguna cosa especial dentro de la UCP. Esta mano la explicaré con más detalle cuando examinemos las instrucciones que la afectan.

Podemos, si queremos, unir las manos B y C, la D y la E, la H y la L para formar nuevos pies, que pasarán a llamarse: BC, DE y HL. Si te fijas en el dibujo, te darás cuenta de que no es posible un pie DL o uno BD o CE, etc. Cada uno de estos nuevos pies es capaz de contar un número con 16 dedos, ya que se comportan, en este sentido, igual que los pies IX e IY.

Los registros

Se suelen llamar registros de la UCP lo que hasta ahora he denominado como pies y manos; así, lo que antes llamábamos la mano A, ahora diremos que es el registro A. El par de pies resultante de unir dos manos se llamará desde ahora par de registros; por ejemplo, lo que antes conocíamos como pies BC, ahora pasará a llamarse par de registros BC. A los pies IY e IX también se les da un nombre especial: registros índice; pero por ahora no te preocupes de ellos, más adelante ya te explicaré el uso que tienen.

El acumulador

Normalmente al registro A se le llama ACUMULADOR porque "acumula" los resultados de las operaciones de la UCP. Puedes pensar que es como la mano derecha de la UCP; date cuenta de que, al igual que muchos de nosotros sólo podemos realizar ciertas tareas con dicha mano, por ejemplo, escribir una carta, etc., la UCP está a veces limitada en algunas de sus operaciones.

El par de registros HL

Este par de registros lo usa a menudo la UCP; es como si fuera un acumulador, pero de 16 dedos; digamos que es ''el pie derecho'' de la UCP.

Registros alternativos

Dentro de la UCP existen más manos que podamos usar, pero sólo para una serie muy limitada de trabajos. Estos registros ''extra'' se llaman registros alternativos y se conocen individualmente como A', B', C', D', E', F', H' y L'.

Lo único para lo que realmente pueden servir es para copiar en ellos, para mayor seguridad, el contenido de las manos principales de la UCP mientras hacemos alguna otra cosa. No podemos hacer sumas con esas manos, sólo usarlas como copias de seguridad de los registros principales.

El puntero de la pila

Este ''pie'' de la UCP señala la dirección de memoria del último elemento que se ha añadido en la pila. Debido a que la pila se va introduciendo dentro de la memoria hacia abajo, el número que señala el puntero disminuye cuanta más información introducimos en la pila. El puntero se actualiza siempre que la UCP introduzca o saque información de la pila.

El contador del programa

Es otro pie de la UCP que le indica la dirección de la siguiente instrucción en lenguaje máquina para que la UCP vaya a buscarla en la memoria y la ejecute. La unidad de control se encarga de «ir a buscar» a la memoria las instrucciones.

La unidad de control

Es como el ''supervisor'' de la UCP; coordina y cronometra el tiempo de las diferentes operaciones de la UCP; además, es responsable, como ya he dicho, de ir a buscar las instrucciones en lenguaje máquina de la memoria. La posición a la que se va a buscar la instrucción se especifica en la dirección que está en el contador del programa. Después, la instrucción se lleva a una de las manos de UCP llamada registro de instrucción.

El registro de instrucción

Este registro de la UCP tiene la instrucción que va a ejecutarse, con lo que la unidad de control sabe de qué instrucción se trata y después la ejecuta.

La unidad aritmético-lógica

Imagínate que es como la calculadora de bolsillo de la UCP, pero en vez de ser tú el que la maneja directamente con tu teclado, a la unidad aritmético-lógica la maneja la unidad de control del procesador.

Sólo está capacitada para hacer sumas y restas; desconoce tanto la multiplicación como la división. También puede comparar números de 8 dedos y ejecutar operaciones con dedos: o bien con los números o con los registros; esto quiere decir que podrás tener dedos estirados o no, según decidas. Los resultados de las operaciones de la ALU también afectan a los dedos que forman el registro *flag*.

Aunque la UCP es un dispositivo muy inteligente, es un inútil sin la ayuda de los otros *chips* del ordenador MSX. Pasemos ahora a ver otros dispositivos que ayudan a que la UCP pueda llevar a cabo sus trabajos u operaciones.

La memoria

El Z-80 puede acceder, gracias al contador del programa de 16 dedos, a 65.535 posiciones diferentes de memoria. Hay dos tipos de memoria dentro del ordenador, que son: la *Read Only Memory* (memoria de sólo lectura), también conocida como ROM, y la *Random Access Memory* (memoria de acceso aleatorio), conocida como la RAM; no te preocupes si no te ha quedado clara la definición de estos términos; ni siquiera es inteligble para los mismos ingleses y americanos. Seguidamente te paso a explicar paso a paso estos dos tipos de memoria.

La ROM

El sistema MSX usa esta memoria para almacenar los programas en lenguaje máquina que forman el sistema operativo del ordenador.

Esta memoria no pierde lo que haya en ella; es decir, su contenido, aunque apagues el ordenador; ni siquiera el programador puede modificar el contenido de la memoria usando el código máquina. Si queremos, podemos cargar los registros de la UCP con números que están en la ROM; obviamente también se pueden ejecutar programas almacenados en la ROM.

La razón por la que a esta memoria se le llama memoria solamente de lectura es porque, por lo anteriormente dicho, es lo único que podemos hacer con ella: leerla.

La RAM

Personalmente, para este tipo de memoria prefiero el nombre no oficial pero sí, a mi juicio, más descriptivo de *Read and Alter Memory* (en español, memoria

legible y modificable), ya que creo describe más correctamente las propiedades fundamentales de esta memoria.

La RAM se usa cuando escribimos programas en BASIC; también la usamos para almacenar en ella nuestros programas en código máquina. Mientras que en este tipo de memoria podemos variar lo que haya almacenado, la RAM tiene una propiedad negativa: siempre que apaguemos el ordenador, la RAM olvida todo lo que se almacenó en ella.

Por este motivo tenemos que guardar nuestros programas en cintas o en discos para tener siempre copias de ellos; lo mismo ocurre con los programas escritos en código máquina.

Como ya he dicho antes, cada posición de memoria sólo puede almacenar un número de 8 dedos; esto sucede tanto en la RAM como en la ROM. El rango de números almacenable en posiciones de memoria irá, por tanto, desde 0 a 255.

Procesador de video

Este dispositivo también conocido como VDP es un *chip* llamado TMS-9928. Se encarga de generar imágenes de televisión bajo el control de la UCP; estudiaremos cómo manejarlo con más detalles en el capítulo 10. El VDP almacena información para producir imágenes en un área especial de la memoria llamada VRAM o RAM de video.

La VRAM tiene propiedades muy similares a la RAM normal que ya expliqué anteriormente, con la diferencia de que ésta se dedica solamente a cubrir las necesidades del VDP.

Generador programable de sonido (PSG)

Como su mismo nombre indica, este dispositivo se encarga de generar tantos sonidos como tu MSX sea capaz de producir. Veremos con más profundidad esta unidad en el capítulo 11.

Interfaz programable para periféricos (PPI)

Aunque este dispositivo tenga un nombre que pueda asustarte, las funciones que normalmente desempeña son muy sencillas. El principal trabajo que a nosotros nos interesa es el de leer el teclado del ordenador que está bajo el control de la UCP. Este *chip* realiza operaciones más complejas, pero por hora y en este libro no las vamos a considerar.

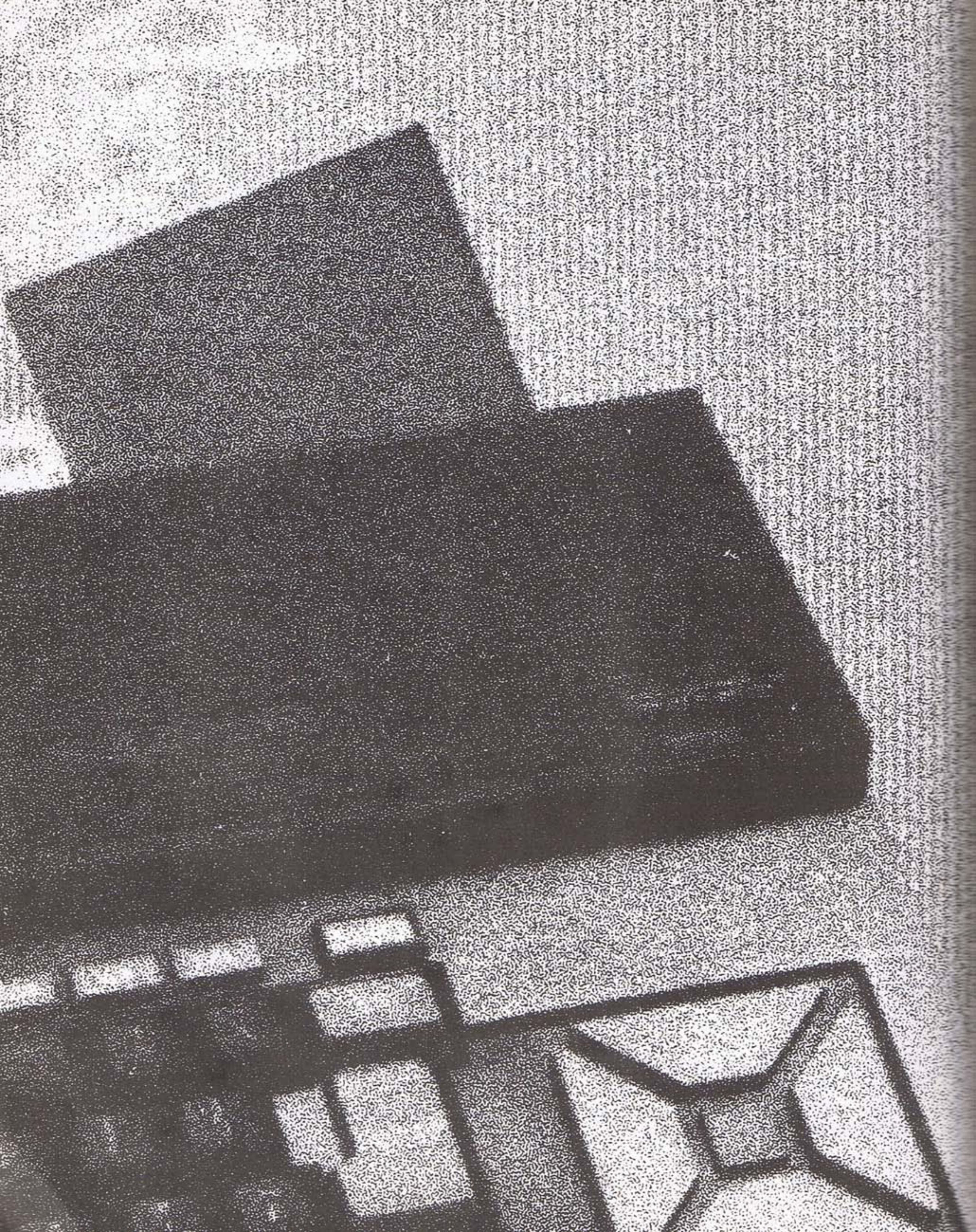
El Z-80 que normalmente está ejecutando el sistema operativo del ordenador es el que maneja el VDP, el PSG y el PPI. Más adelante veremos cómo manejar dichos dispositivos con tus programas en lenguaje máquina. Puede que todo lo que te aconsejo que hagas no seas todavía capaz de hacerlo, pero no debes preocu-

parte, porque es mucho más fácil de lo que te imaginas; de hecho, ya has visto un pequeño programa que controla el VDP.

Hasta ahora lo que he hecho es explicar someramente la mayoría de dispositivos que se verán con mayor profundidad a lo largo del libro; ahora veremos cómo programar otros dispositivos del sistema. Y digo programar ya, aunque todo ese tipo de operaciones las haga la UCP; el VDP, por ejemplo, tiene manos que controlan sus propias operaciones, por lo que, si modificases los valores que hay en los registros del VDP, alterarías la conducta del VDP y, por tanto, también la imagen que apareciera en la pantalla.

Los números se mandan a los registros en el VDP o *chip* del sonido desde los registros de la UCP, y el ''arte'' de programar el VDP y la PSG para que realicen determinadas tareas depende de los números que enviemos desde la UCP a dichos dispositivos, como veremos más adelante.

Pasemos ahora a estudiar una operación fundamental para los ordenadores: exactamente, ¿cómo cuentan ellos?



2

Cómo cuentan los ordenadores

Ya dije en el capítulo 1 que la UCP puede representar números entre 0 y 255 con cada una de sus manos de 8 dedos, pero ¿cómo puede hacer esto la UCP si nosotros con sólo 10 dedos sólo podemos contar hasta 10?

Pues bien, cuando nosotros contamos con nuestros dedos, damos a cada dedo un mismo valor, por ejemplo: cada dedo que estiramos representa el mismo valor que otro que está a su lado, estirado también; pero, si lo piensas un poco, no es la única forma de contar con los dedos; me explico: de hecho, tú podrías usar dedos diferentes para diferentes valores; por ejemplo: el primer dedo que estiramos para contar podría representar el valor 1; el dedo contiguo haremos que valga, por ejemplo, 2. Desde luego, cuando cerramos la mano y bajamos dichos dedos ambos valdrán 0; por tanto, puedes ver que si representas el valor "1" con el dedo índice y luego el valor "2" para el dedo corazón, al sumarlos, lo que obtendrías sería un 3, es decir, 2 + 1 = 3.

Como es obvio, este método es más eficiente que el nuestro, ya que con el nuestro necesitamos tres dedos para representar el número 3, mientras que con este sólo necesitarías dos dedos. En el método de contar que usa la UCP se observa que:

1. Tanto si el dedo está levantado como si no, lo importante es el número que dicho dedo representa.
2. La posición del dedo en la mano determina el valor que ese dedo representa y, por tanto, también determina el número representado por la mano entera.

Pasemos a ver este método de representación de números, usando dos dedos:

		Número representado
	=	0
	=	1
	=	2
	=	3

Vamos a representar un dedo levantado por el dígito "1" y uno bajado por el dígito "0". Una vez transformado el dibujo de arriba, queda:

0000	=	0
0001	=	1
0010	=	2
0011	=	3

Estoy seguro de que esta notación ya debe serte familiar. ¿Recuerdas nuestro método de representar la presencia o ausencia de una señal eléctrica? Aquí también usamos ceros y unos. Este método de representar números con sólo dos estados diferentes (el dedo levantado o bajado, señal o no señal, un 1 o un 0) se denomina REPRESENTACION BINARIA de un número. Si sumamos un dedo al método anterior, que constaba sólo de dos, obtendremos siete combinaciones diferentes de dedos; si lo dudas, no tienes más que comprobarlo tú mismo con tus dedos. Ahora ya puedes representar números del 0 al 7, con tres dedos. Usaremos el "0" y el "1" para representar los números en vez de los dedos; recuerda que el "0" es un dedo bajado y el "1" un dedo levantado.

000	=	0
001	=	1
010	=	2
011	=	3
100	=	4
101	=	5
110	=	6
111	=	7

Si sumamos otro dedo más, serían ya cuatro, con lo que podríamos representar números del 0 al 15. En informática, los números del 10 al 15 se representan por letras de la A a la F en vez de con dos números; su representación es como sigue:

10 = A
11 = B
12 = C
13 = D
14 = E
15 = F

A este método de representar números se le llama hexadecimal; la representación de los números del 0 al 15 es en este método como sigue:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

En los ordenadores MSX puedes usar números hexadecimales, aunque estés en lenguaje BASIC. Sin embargo, es necesario decirle al ordenador cómo distinguir los números hexadecimales de los decimales.

En el MSX a los números decimales se les asigna el prefijo "&H"; en otros ordenadores también se les puede asignar los prefijos "&" y "H", pero el MSX sólo entiende la notación hexadecimal con el prefijo "&H"; por esta razón, puede que te encuentres otros libros en los que para designar el número 10 en hexadecimal se utilizan notaciones diferentes:

&A = 10
&HA = 10
HA = 10

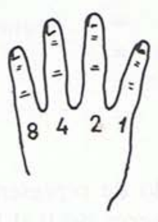
Otra notación diferente es la que utiliza el sufijo "H" para representar números hexadecimales; por ejemplo, para el número 10 sería AH. En este libro se utilizará el prefijo &H.

Como verás, es muy conveniente representar las instrucciones que estén en código máquina, en hexadecimal: por ejemplo, 4 dedos pueden representarse con un solo carácter, y, como ya te habrás dado cuenta, el número formado por una mano de 8 dedos puede representarse sólo usando dos caracteres. Las ventajas de representar las instrucciones en código máquina como instrucciones en hexadecimal son las siguientes:

1. Es muy fácil pasar números hexadecimales a sus correspondientes en binario, con lo que podemos ver qué dedos están levantados o no en el número.
2. Podemos saber, por el número de dígitos del número, si cabrá en una o dos manos o no; los números representados por una mano tienen 2 dígitos y los números representados por dos manos tienen 4 dígitos.

¿Cómo podemos saber el valor que representa un número de 4 dedos en binario?

Como ya debes saber, cada dedo tiene un valor determinado; por ejemplo:



Tenemos 4 dedos levantados y he asignado un valor a cada dedo. Cuando, por ejemplo, se levanta el dedo último, tiene el valor 8, y cuando lo bajamos, el valor 0. Si todos los dedos están estirados, tenemos que el valor total que representa es la suma de todos, es decir:

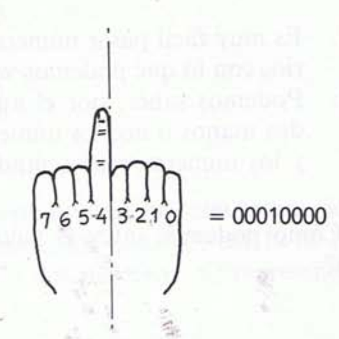
$8 + 4 + 2 + 1 = 15$

Simplemente se suman los valores de aquellos dedos que se han levantado. Si tu mente es algo matemática, habrás notado ya que el valor que representa cada número es el anterior pero multiplicado por 2 (de derecha a izquierda). Si numeramos los dedos de la siguiente forma:

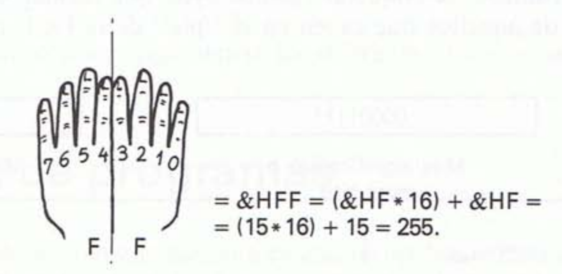


el valor asignado a cada dedo es 2 elevado a *n*, donde *n* es el número del dedo: por ejemplo, 2 elevado a 0 sería 1.

Hasta aquí hemos visto cómo representar números de 0 15; veamos ahora qué deberemos hacer para representar números superiores a 15: sencillamente añadiendo dedos a la mano. Así llegaremos hasta formar números de 8 dedos que la UCP representa en sus registros. Por ejemplo, el número 16 se representaría con el cuarto dedo levantado:



Este número en hexadecimal es: &H10. Lo que hacemos es asignar un dígito hexadecimal a cada parte de 4 dedos. En el ejemplo anterior el trozo de mano de 4 dedos a la derecha representa un 0, y los 4 dedos a la izquierda representan un 1. Sin embargo, estos dos fragmentos de manos no representan el mismo valor real, es decir, a la hora de hallar el número total, el valor del trozo de mano izquierda representa 16 veces más que el valor de la mano derecha; fíjate en el ejemplo siguiente con 8 dedos levantados:



Con este ejemplo ves cómo se representa el número 255 con 8 dedos, y a la vez cómo la UCP cuenta hasta 255 con sus manos de 8 dedos. De forma muy parecida es como se representan, con 16 dedos, números hasta 65535.

Los bits y los bytes

Es momento ya de introducir los nombres informáticos de lo que hasta aquí hemos llamado "dedos" y "manos".

En español a veces se llama dígitos a los dedos. Este nombre es más apropiado para nombrar los ceros y unos en el código binario; por tanto, y desde ahora, debemos llamar números de 8 dígitos a los números de 8 dedos. Hay aún otro nombre para los dígitos binarios; dicho nombre en términos informáticos es: bit.

Un bit es la abreviatura de *Binary digit* (dígito binario, en español); en informática a los números de 8 dedos se les llama números de 8 bits y al conjunto de esos 8 bits se les llama BYTE; un BYTE es, por tanto, un número que puede ser representado por 8 dedos o también un número que cabe en una mano de la UCP, o también un registro. Las "fracciones" de manos también poseen un nombre digamos más informático: *nibble* (*nibble* significa mordisco pequeño, en inglés) y *bite* (con i latina) quiere decir mordisco; o sea, que "mano" es a "fracción de mano", como "byte" es a *"nibble",* aunque en realidad no sería byte, sino *bite*.

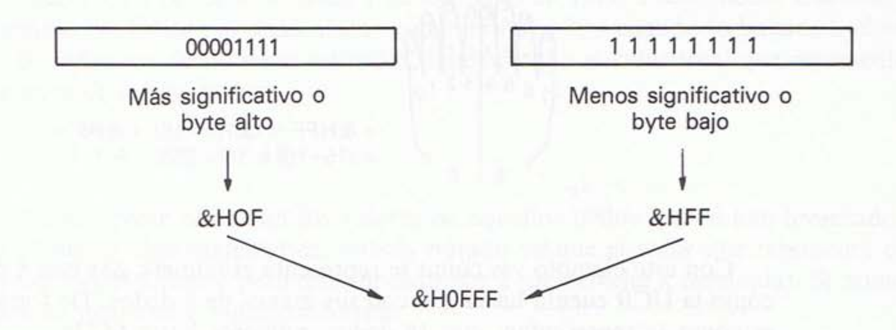
En informática son ya muy usados los términos bit, byte y *nibble,* por lo que, si no estás ya muy acostumbrado a oírlos, cuando acabes este libro te aseguro que pasarán a formar parte de tu vocabulario no habitual, pero sí informático.

Al igual que numeramos los dedos (al primero, pulgar; al segundo, índice, etcétera), también vamos a numerar los bits y los bytes; el sistema de numeración

es el mismo que utilizábamos para la mano de 8 dedos de antes, la que iba de 0 a 7 y de derecha a izquierda.

Al bit 0 también se le conoce como bit menos significativo del byte o con la sigla inglesa LSB *(Least Significant Bit);* este bit es el menos significativo, ya que su valor asociado, el 1, es el que menos contribuye al valor del número representado por los 8 bits. Al bit 7 se le llama bit más significativo del byte o MSB en inglés *(Most Significant Bit).*

También se etiquetan los dos bytes que forman un número de 16 dedos, es decir, de aquellos que caben en el "pie" de la UCP o en un par de registros.



En este ejemplo el byte alto es 256 veces más significativo que el byte bajo. De esta manera el valor total en el número de 16 bits viene dado por:

valor = 256 * ALTO + BAJO

donde BAJO es el valor del byte bajo. De forma parecida, un par de registros de 16 bits en la UCP, como por ejemplo el par HL, o un pie de la UCP, como por ejemplo el registro IX, están formados por registros altos y bajos.

El registro llamado alto tiene, por tanto, la mayor porción del número, y el segundo tipo de registros, la menor parte del número.

Por ejemplo, en el par de registros BC, B tiene el byte alto, y C el byte bajo. Esta notación no es difícil de recordar; si tomas como referencia el par de registros HL, recuerda que la H viene de HIGH (alto) y L es de LOW (bajo). Este es probablemente el porqué de no usar G cuando se puso nombre a los registros. Los registros GH hubieran sido una total confusión.

Representación de la información

Los seres humanos trabajamos con toda clase de información; dicha información está normalmente en forma de letras y números; sin embargo, los ordenadores sólo tratan, realmente, con números. Es lógico pensar que el ordenador debe

tener alguna forma de representar otros tipos de información. Hay dos tipos de información codificados numéricamente en la memoria del ordenador:

1. Programas en lenguaje máquina; por ejemplo, un programa nuestro, o también puede ser el sistema operativo del ordenador.
2. Los datos para los programas en lenguaje máquina, que pueden ser tanto números como letras. Así un programa BASIC por ejemplo puede ser definido como datos para que el sistema operativo trabaje con ellos.

Veamos ahora cómo se representan los diferentes tipos de datos en la memoria del ordenador.

Representación de programas

Un programa en lenguaje máquina es una secuencia de instrucciones almacenada en la memoria del ordenador. Las instrucciones Z-80 pueden ocupar de uno a cuatro bytes; por ejemplo: la orden HALT es una instrucción de un solo byte.

Una vez que un número cualquiera que represente una instrucción se ha extraído de la memoria y ha sido ejecutado por la UCP, el ordenador buscará la instrucción siguiente; si la UCP se da cuenta de que necesita más de un byte para formar una instrucción completa, buscará bytes en la siguiente posición hasta que complete la instrucción.

Es obvio, por tanto, que una instrucción de un solo byte se ejecute normalmente más rápidamente que otras instrucciones más largas.

Los datos

En BASIC se utilizan diferentes tipos de variables para almacenar información, de manera que nuestro programa pueda trabajar con ellos.

Estas variables son:

NUMEROS ENTEROS
NUMEROS REALES
CADENAS DE CARACTERES

En código máquina no ocurre lo mismo, ya que los únicos números con los que la UCP puede trabajar son los enteros dentro del rango 0 a 255 o bien los que van desde el 0 al 65535, como ya hemos visto.

Para que la UCP pueda usar números reales, tiene que ser programada para ello, pero se necesitan tantos programas para lograr que la UCP trabaje con esos números, que normalmente es mejor resolver los programas con números reales en BASIC que en código máquina. Las cadenas alfanuméricas de caracteres son más fáciles de usar por la UCP; las letras que forman la cadena están representadas en el ordenador por números.

Los enteros

Hemos visto ya que la UCP trabaja con números reales siempre y cuando dichos números vayan del 0 al 65535. Es muy lógico preguntarse: ¿y qué pasa con los números negativos?, ¿cómo se representan los enteros con signo, si es verdad que el ordenador trabaja realmente con ellos?

Los enteros con signo

Para la UCP, estos números es mejor representarlos en binario, así que necesitaremos un método de representar el signo de un número, algo parecido a lo que normalmente en aritmética se llama el "—" y el "+".

La forma más común de representar el signo negativo es levantando el pulgar de la mano, es decir, al bit más significativo se le asigna un 1; esto además indica que, para un número de 8 bits, los bits del 0 al 6 representan el valor real del número y al bit 7 se le llama bit de signo de un número de 8 bits.

Es lógico pensar que, puesto que ahora sólo tenemos 7 bits para representar un número, no se podrán representar números cercanos al 255; pero, por el contrario, date cuenta de que, al haber ahora números negativos en el ordenador, la mitad de los números que representa dicho byte serán negativos, o lo que es lo mismo: al bit 7 se le asigna un 1, y la otra mitad de números serán positivos, es decir, el bit más significativo será un 0, con lo que obtendremos un rango que va desde —128 a +127 en un solo byte. Esto crea un problema importante: ¿cómo sabremos cuándo un byte representa un número grande positivo o negativo? La respuesta está en la interpretación que cada programador esté usando; si, por ejemplo, vas a usar números con signo, ya sabes que el bit 7 es el bit de signo y, por tanto, no representará ningún valor digamos numérico. De un modo o de otro, las instrucciones en código máquina funcionarán correctamente, pero obviamente el resultado de cada operación dependerá de si usas números con signo o no.

Crear números negativos no es sólo cuestión de asignar un 1 al bit 7. Ahora tendremos que resolver qué combinación de bits de 0 a 6 del número representa el número en cuestión.

Lo más importante que debes recordar sobre los números negativos es que cuando los sumas a sus correspondientes números positivos el resultado debe ser 0; ejemplo:

—1 + +1 = 0

Como ves, la representación del —1 debe ser aquella a la que al sumar +1 el resultado debe ser 0; así:

```
   00000001
+  1???????
   --------
   00000000  RESULTADO DESEADO
```

Si representamos — 1 con el número binario 10000001, tendremos que el bit 0 sería un 0:

$$
\begin{array}{r} 00000001 \\ +\ 10000001 \\ \hline 10000010 \end{array}
$$

Sin embargo, ésta no es la respuesta que buscábamos; tendremos que tomar el acarreo que se generó con la suma que hicimos en el bit 0 y hacer que el resto de los bits se queden a 0; para esto tenemos que dar el valor 1 a los bits de la combinación binaria de — 1:

$$
\begin{array}{r} 00000001 \\ +\ 11111111 \\ \hline 00000000 \end{array}
$$

Como ves, éste sí es el resultado que esperábamos para el número en binario — 1, pero ¿funcionará esto con otros números negativos? La mejor manera de saberlo es aplicar este método a otros números.

Antes de empezar a ver lo dicho anteriormente, tendremos que intentar idear una regla general para conseguir representar en binario cualquier número negativo.

La combinación — 1 anterior se consiguió en dos fases: en la primera, cambiamos todos los ceros de la combinación positiva por unos y todos los unos por ceros; es decir:

00000001 = 11111110

A esto se le llama COMPLEMENTAR un número; el complemento de 1 es 0, y el de 0 es 1. La segunda fase fue simplemente sumar 1 al complemento del número:

1111110 + 00000001 = 11111111

Vamos a ver si efectivamente nuestra regla funciona para todos los casos aplicándolo a otros números como dije antes; veamos si somos capaces de obtener la combinación binaria de — 2:

1. Escribamos la combinación binaria + 2:

   00000010

2. Hallemos el complemento del número anterior:

   11111101

3. Sumemos 1 al complemento obtenido:

   11111101 + 00000001 = 11111110

Este último resultado tendría que ser la combinación binaria de —2; veamos si es cierto sumándole +2, y si el resultado es 0, nuestro método FUNCIONA.

```
  11111110
+ 00000010
  --------
  00000000
```

¡SI!, volvió a dar 0; este método de representar números negativos en binario se llama representación en COMPLEMENTO A DOS y es el método más conocido y usado para representar dichos números. Si aplicas la misma técnica a un número negativo, lo que obtendrías es el mismo, pero positivo; esto es lógico, ya que, si recuerdas, la suma de dos números negativos es una suma.

Hasta ahora hemos visto esta técnica para los números de 8 bits, pero también sirven para los de 16.

Cuando usamos la notación en complemento a dos para representar un número negativo de 16 bits, el bit 7 del byte mayor es el bit de signo, mientras que el bit 7 del byte menor no representa ya el signo del número. Cuando usamos números de 16 bits para representar números negativos, el rango de números que obtenemos va desde —32768 a +32767, en vez de ir desde 0 a 65535.

Conviene señalar que las variables enteras del MSX BASIC, aquellas cuyo nombre va seguido del signo "%", se almacenan en la memoria del ordenador como valores de 2 bytes con signo y, por tanto, tienen valores que van desde —32768 a +32767. Si quieres más detalles sobre cómo se almacenan las diferentes variables MSX en la memoria del ordenador, puedes verlo en los apéndices del libro.

Caracteres y cadenas

En algunos programas en código máquina puede que, en vez de querer que los números representen instrucciones en lenguaje máquina, deseemos representar también con números algunos caracteres.

Antes de seguir con esta explicación, es conveniente que definamos qué es un carácter. Un carácter es algo que se puede escribir en la pantalla usando la instrucción PRINT del BASIC. Así, el conjunto de letras y números del mensaje "123 HOLA" son todos caracteres, o también se dice que son una cadena de caracteres o, más sencillamente, una cadena. Como no hay más de 255 caracteres disponibles para el programador, los números que se usan para representar caracteres son todos de 8 bits. En MSX, el número que se usa para representar un determinado carácter se llama código ASCII del carácter.

ASCII es la sigla de *American Standard Code for Information Interchange,* que en español es lo mismo que Código Estándar Americano para el Intercambio de Información. Por ejemplo, el código para la letra "A" es el 65. Para saber qué clave ASCII tiene un determinado carácter, se usa la función del BASIC ASC( ); por ejemplo:

```
PRINT ASC ("!")
```

que nos devolverá un 33; luego ya sabemos que el código ASCII para "!" es 33.

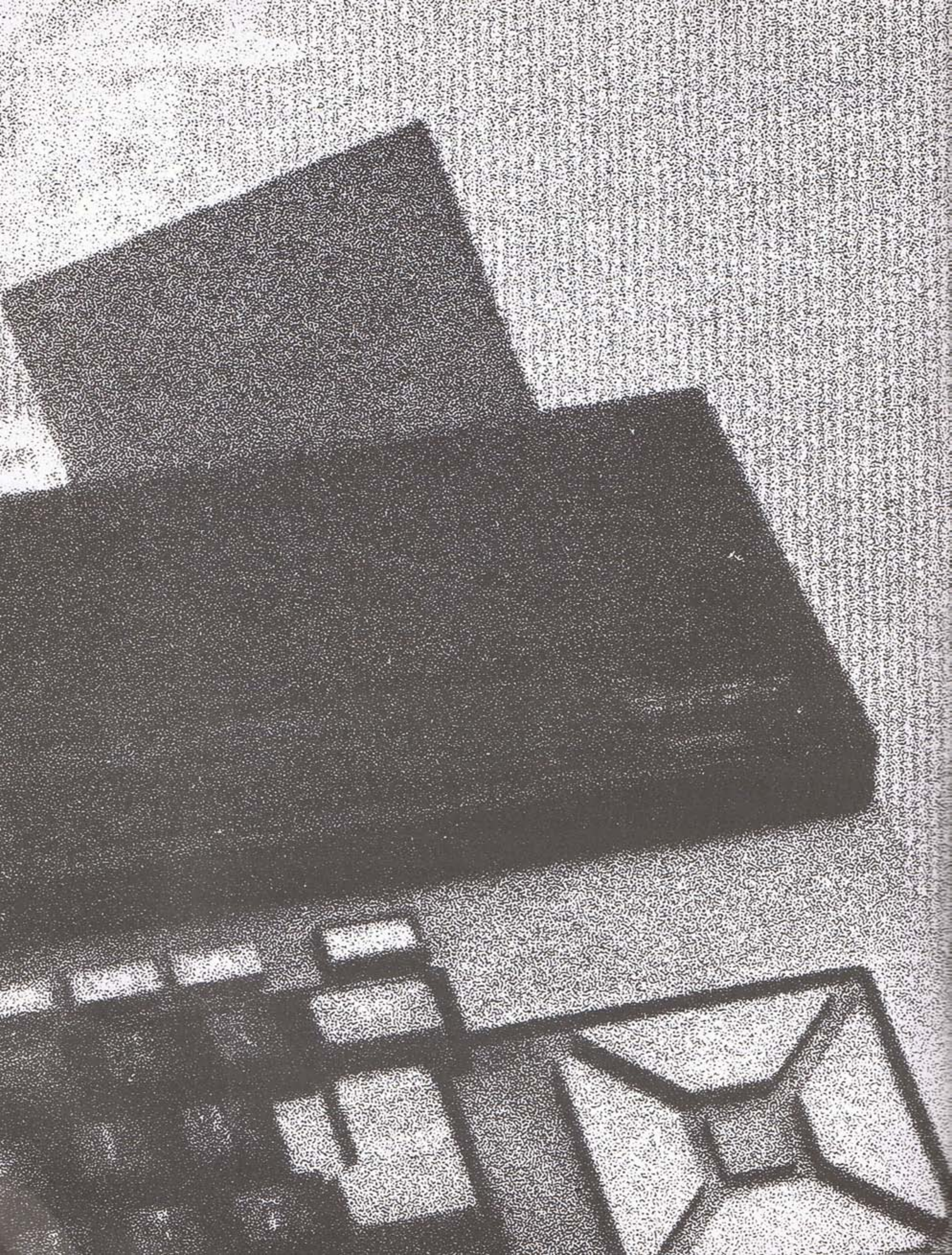
Resumen

Como ya te habrás dado cuenta, lo que un determinado número representa en la memoria del ordenador depende, principalmente, de lo que el programador quiere que represente; así un número puede representar lo siguiente:

1. Una instrucción en lenguaje máquina.
2. Un número cuyo rango va de 0 a 255.
3. Un número comprendido entre — 128 y + 127.
4. Una parte de un número de 16 bits.
5. Un carácter alfanumérico.

Es importante que el programador tenga en cuenta para lo que usa las diferentes partes de la memoria del ordenador; sería un desastre para la UCP tratar una secuencia de bytes que representa una frase del texto como un programa.

El problema es saber cómo introducir dichos números dentro de la memoria de tu ordenador, tanto si son datos o instrucciones en código máquina, y cómo conseguir que la UCP ejecute programas en código máquina que hayas escrito. Como es lógico, necesitaremos del BASIC para conseguir lo anterior. En el siguiente capítulo te explicaré diferentes instrucciones del BASIC que nos serán útiles para nuestros programas en lenguaje máquina.



3 Código máquina frente al BASIC

Como ya dije antes, cuando encendemos nuestro ordenador MSX, la UCP Z-80 ejecuta el programa intérprete BASIC que está almacenado en la ROM del MSX. Cuando queramos decirle a la UCP que ejecute los programas en código máquina que nosotros hayamos escrito, tendremos que ordenar a la UCP que abandone por un momento el programa de la ROM y vaya a ejecutar el nuestro. Esto se lo indicaremos gracias al BASIC, ya que es el programa intérprete de BASIC el que normalmente se está ejecutando.

En este capítulo vamos a ver a lo que yo llamo el "interfaz" BASIC entre los programas en lenguaje máquina que queremos introducir y ejecutar en la memoria del ordenador y el lenguaje BASIC que nos ayudará a introducir y ejecutar dichos programas.

Te explicaré algunas instrucciones en BASIC que son de gran utilidad a la hora de escribir programas en lenguaje máquina; también veremos cómo pasar información entre los programas en BASIC y los que están en lenguaje máquina. Este último punto es muy importante, ya que queremos, con bastante frecuencia, escribir pequeños programas en lenguaje máquina que hagan cosas que el BASIC no podría hacer.

Dónde poner nuestros programas en código máquina

Lo primero que deberemos hacer cuando vayamos a escribir programas en código máquina es buscar un sitio en la memoria del MSX para ponerlo. Es lógico

que tiene que ser en la RAM; no olvides que no podemos modificar lo que hay en la ROM, por lo que tampoco podremos introducir bytes en ella.

Algunas zonas de la RAM del MSX no están realmente disponibles para almacenar los bytes que formarían los programas en lenguaje máquina. Estas zonas de memoria son aquellas usadas o bien por el programa en BASIC o bien son las que usa temporalmente la UCP Z-80 mientras ejecuta el programa intérprete BASIC. Por tanto, cualquier programa en código máquina que esté en dichas zonas de memoria puede ser borrado por el ordenador mientras ejecuta el intérprete o mientras tú introduces líneas de programas en BASIC. La zona de memoria que se usa para almacenar las variables utilizadas por el programa en BASIC se denomina área de trabajo del programa BASIC.

El área de memoria que usa temporalmente la UCP se llama zona del trabajo del sistema. Si cambiamos cualquiera de los bytes de dicha zona, es bastante probable que cambiemos también parte de la información que la UCP necesita para ejecutar el intérprete del BASIC. No es sorprendente, pues, que el Z-80 pierda "la pista" de lo que está haciendo y pierdas el control del ordenador. Se puede decir que el ordenador se "cuelga", y lo único que podrás hacer para recuperar el control es pulsar la tecla RESET. Probablemente ya sabrás que, si pulsas dicha tecla, pierdes para siempre todo lo que hubiera en la RAM.

La moraleja que debes sacar de esta, digamos, fábula es que, cuando estés experimentando con programas en código máquina, debes guardar todos los bytes que forman los programas en cassettes o discos antes de intentar ejecutar dichos programas; de esta forma, si pierdes el control del ordenador, lo único que tendrás que hacer sería cargar otra vez los bytes de la cinta o del disco en el ordenador. Como es lógico, esto es mucho más fácil y rápido que teclearlo todo otra vez.

Cuando comiences a programar en código máquina, es muy probable que pierdas el control del ordenador a menudo, así que estáte muy preparado por si esto sucede. Esto ocurre porque los programas en código máquina son implacables con los errores que se cometan en la programación. En BASIC al programador se le facilita ayuda con mensajes de error que le dan la oportunidad de cambiar los despistes que ha tenido en el programa.

Si tienes suerte, puede que sólo tengas un pequeño susto; pero si tienes mala suerte (lo cual es mucho más probable), el ordenador se volverá loco y te enterarás de lo terrible que es este tipo de accidentes.

De todas formas, volvamos al problema de cómo encontrar un sitio para poner nuestros programas en código máquina. Obviamente, todo espacio que usemos es mejor que lo pongas a salvo de ser escrito por el BASIC o por la memoria ROM del MSX. La forma más segura de hacer esto es reservar una parte de memoria en la zona de trabajo del BASIC con la instrucción CLEAR. Como sabes, la instrucción CLEAR normalmente borra todas las variables en BASIC de nuestros programas. Esta nueva instrucción CLEAR guarda un área de memoria entre la zona de trabajo del intérprete de BASIC y la zona de trabajo del sistema, que está a salvo de ser modificada por accidente por el intérprete de BASIC.

La nueva opción de la instrucción CLEAR es como sigue:

```
CLEAR n,m
```

donde n y m son números enteros. De la n no hace falta que nos ocupemos; normalmente yo la fijo a 200. El número m es el más importante para nosotros, que es la (dirección — 1) de la primera posición de memoria que vamos a reservar para programas en lenguaje máquina. La cantidad de memoria que se guardará serán todos los bytes que hay entre esta dirección y el principio de la zona de trabajo del sistema; normalmente ésta es la memoria que hay entre el parámetro $m + 1$ y la dirección 62334 inclusive, ya que la zona de trabajo del sistema empieza en 62335.

Ahora veremos cómo utilizar dicha instrucción, ya que, como bien imaginas, es muy importante. El espacio reservado estará entre los programas BASIC y las variables, y la zona de trabajo del sistema; con esto disminuye la cantidad de memoria que está disponible para programas en BASIC y variables.

Ya que estamos ocupando memoria de la zona del BASIC, lo primero que tendremos que hacer es hallar la dirección del último byte que normalmente esté disponible para los programas en BASIC. Dicho byte será siempre el mismo para tu ordenador. Después de todo, el intérprete BASIC siempre te dará la misma cantidad de espacio para tus programas, aunque esto pueda variar en los distintos ordenadores MSX. Podemos hallar dicha dirección de la siguiente manera:

Primero hallaremos la dirección del primer byte de memoria que usa el BASIC como zona de trabajo. Cuando compres tu ordenador, la cantidad de RAM que tiene puede ser de 16K o de 32K. Esta memoria se divide entre los programas BASIC y la zona de trabajo del sistema. Si tu ordenador tiene más de 32K de memoria, actúa como si tuvieras 32K, ya que en este caso no importa.

Por si no sabes lo que es un "K", te diré que es una abreviación informática que equivale a 1.024 bytes; 2K, por tanto, serán 2.048 bytes; 12K será 512 bytes; 16K son 16.384, etc. Si pasas el valor de los "K" a bytes y luego lo restas al número 65536, obtendrás el comienzo de la RAM. Por ejemplo, en un ordenador de 16K, el comienzo de la memoria se halla así:

comienzo = 65536 — (16 * 1024)

El resultado es 49152; ahora tenemos que hallar la dirección del último byte de la zona de trabajo del BASIC. Existe una instrucción en BASIC llamada FRE (0), que te permite hallar qué espacio queda en la zona de trabajo del BASIC. La instrucción PRINT FRE (0) mostrará en la pantalla la memoria que queda para los programas en BASIC y las variables. Si pulsas la tecla RESET y borras todos los programas del ordenador, al ejecutar la instrucción PRINT FRE (0) te saldrá en pantalla la memoria que está disponible para programas en BASIC.

Así, ahora podemos hallar la dirección de memoria del último byte usado como zona de trabajo del BASIC. En un ordenador de 16K la dirección de memoria donde comienza el BASIC es la dirección 49152; el último byte de memoria viene dado por:

49152 + FRE (0)

que da 61583 en un ordenador de 16K; lo que hemos calculado es la dirección del último byte de memoria disponible para el BASIC. Ahora tendremos que guardar

o reservar espacio para nuestros programas en código máquina. Por ejemplo, suponte que quieres guardar 200 bytes de memoria para tus programas en código máquina:

Lo primero que debes hacer es restar dicho número de la dirección que indica el final reservado para los programas en BASIC, que hemos ya hallado más arriba, es decir:

61583 — 200

que es igual a 61383, que a su vez es la dirección del primer byte de memoria que está disponible para nuestros programas en código máquina. Ahora necesitamos la instrucción CLEAR para reservar esta zona de memoria para que quede a salvo de los estragos del BASIC. Recuerda que establecimos el parámetro "*m*" de esa instrucción para que tuviera el valor de la (dirección — 1).

La instrucción en este caso es:

```
CLEAR 200 , 61382
```

El espacio que hay entre 61383 y 61583 está ahora a salvo de nuestros programas en código máquina. El intérprete BASIC sólo podrá usar la memoria que va hasta la dirección 61382 para guardar programas en BASIC y variables.

Hay una gran parte de RAM del ordenador que todavía no he mencionado como posible lugar para programas en código máquina; me refiero al procesador de video (PV, en español, y VDP, en inglés —*Video Display Processor*—), que almacena la información sobre las imágenes que van a salir en la pantalla; pero, desafortunadamente, no hay manera de poder usar esta memoria para almacenar programas en código máquina, debido a que la UCP no puede acceder directamente a la zona de memoria RAM para video o VRAM sin utilizar el procesador de video (PV). El control del PV lo lleva un programa en código máquina.

Como verás, sería imposible ejecutar un programa en lenguaje máquina que necesitase de otro para extraer los bytes que forman el primer programa. Además, la instrucción SCREEN y las instrucciones del MSX que modifican el tipo de pantalla también modifican el contenido de la zona de memoria para video (VRAM).

Ahora ya eres capaz de reservar espacio para los bytes que forman tus programas en código máquina, pero, realmente, ¿cómo introducimos esos bytes dentro de la memoria?, y ¿podemos acceder al contenido de una posición de memoria? Estas dos cuestiones se contestan gracias a dos instrucciones del BASIC llamadas POKE y PEEK.

Las instrucciones POKE y PEEK

Para introducir bytes en una memoria se usa la instrucción POKE. Lo que hace exactamente es introducir o asignar bytes en posiciones de memoria determinadas. La instrucción tiene el siguiente formato:

POKE dirección, valor

donde ''dirección'' es la dirección de la posición de memoria que queremos cambiar y ''valor'' es el valor o número del byte que vamos a asignar a dicha dirección. Aunque puedes asignar valores en cualquier dirección del ordenador, ten en cuenta que es inútil que asignes valores a la ROM, ya que no podrás escribir nada en ella.

Ten cuidado también de no alterar los bytes de la zona de trabajo del sistema cuando asignes bytes en la RAM, ya que podrías perder el control del ordenador. Si la dirección en la que estés introduciendo un byte ya tuviera otro que formase parte del programa en BASIC, seguramente lo habrás dejado ininteligible para el intérprete BASIC.

Una última advertencia: no puedes asignar nada en la VRAM con la instrucción POKE; hay otra instrucción para la VRAM que hace esto y que la veremos más adelante.

Si hemos usado la instrucción POKE para asignar en la memoria una secuencia de bytes que son nuestro programa en lenguaje máquina, creo que sería conveniente mirar en las posiciones de memoria pertinentes si todos los bytes están correctamente escritos o asignados. Eso se hace con la instrucción PEEK, que nos permite coger posiciones de memoria y ver lo que hay en ellas. La instrucción se utiliza así:

```
PRINT PEEK(dirección)
```

o

```
LET A = PEEK(dirección)
```

donde ''dirección'' es la dirección de la posición de memoria que queremos ver. El primer ejemplo escribirá el contenido de la posición de memoria en la pantalla y el segundo dará a la variable en BASIC ''A'' el valor del contenido anterior.

En todas las instrucciones PEEK y POKE, los parámetros que se usan son números enteros.

Estas dos instrucciones son las más importantes, en BASIC, a la hora de poner en memoria programas en código máquina y nos permiten modificar el contenido de las posiciones de memoria y además conocer exactamente el valor de determinadas posiciones de memoria.

Una vez que tengamos en la memoria del ordenador una secuencia de bytes que formen un programa, para que éste haga algo necesitamos ejecutar el programa. No pienses que esto se hace sólo tecleando RUN o GOTO 61383; estas instrucciones sólo ejecutarán o bien un programa en BASIC o generarán un mensaje de error; esto se debe a que la UCP Z-80 está aún ejecutando el intérprete de BASIC. El problema principal que presenta ejecutar otro programa en lenguaje máquina que no sea el intérprete es sencillamente que tienes que conseguir que la UCP deje el intérprete y empiece a ejecutar las instrucciones que encuentra en otras direcciones del ordenador. Afortunadamente, hay dos instrucciones en BASIC que nos permiten hacer eso.

Las instrucciones DEFUSR y USR

La instrucción que usamos para decirle a la UCP que deje el programa intérprete BASIC y que ejecute una parte de código máquina de una determinada dirección se llama USR. Se pronuncia *user* y se refiere al código máquina definido por el usuario.

Tenemos a nuestra disposición diez instrucciones USR diferentes numeradas de 0 a 9 y todas se encargan de ordenar a la UCP que vaya a ejecutar el fragmento de código máquina que hay en una determinada dirección.

La instrucción USR0 nos indica un fragmento en código máquina en una dirección concreta; la USR2 se refiere a otra, etc. Sin embargo, antes de poder usar dichas instrucciones para ejecutar fragmentos de lenguaje máquina, tenemos que indicar a cada instrucción USR dónde están en el ordenador los fragmentos de código máquina que se van a ejecutar; para esto necesitamos la instrucción DEFUSR, abreviatura de *DEFine USR* (en español significa definición de la USR). Con ésta instrucción podremos indicar a cada instrucción USR la dirección del programa en código máquina que tiene que ejecutarse; por ejemplo, la DEFUSR1 le indica a la rutina USR1 dónde puede encontrar su correspondiente programa en lenguaje máquina; la DEFUSR2 se lo indica a la instrucción USR2 y así sucesivamente, excepto en el caso de la USR0 y la DEFUSR0, donde podemos no poner el 0, porque se sobreentiende, pero ¿cómo se usan dichas instrucciones? Lo primero que deberemos hacer es asignar a la dirección del programa en código máquina la instrucción DEFUSR. Suponte que tenemos un programa en la dirección 61390 y queremos ejecutarlo con la instrucción USR1; tendremos, por tanto, que usar la instrucción DEFUSR1 de la siguiente manera:

DEFUSR1 = 61390

Como ves, esto es muy fácil; sencillamente escribes la dirección de la primera instrucción del programa en lenguaje máquina y después el signo "=". Para ejecutar el programa en esa dirección se llama a la instrucción USR:

PRINT USR1 (0)

ahora ejecutamos el programa en lenguaje máquina en la dirección 61390. De forma parecida la instrucción DEFUSR = 60000 indica la dirección donde comienza el programa en código máquina que será llamado por la instrucción USR0.

Fijémonos más detenidamente en la instrucción USR. El valor entre paréntesis tiene dos nombres distintos: se le puede llamar parámetro o argumento. Este puede ser un entero, una cadena de caracteres o un número real. El argumento puede ser o bien una variable, como por ejemplo, A%, A$ o A, o bien una constante, como 1, "FRED" o 1.2345. Teniendo en cuenta todas las instrucciones DEFUSR correspondientes, son correctas las siguientes instrucciones USR:

```
PRINT USR(3)
PRINT USR(A%)
PRINT USR(1.2345)
PRINT USR(A$)
PRINT USR("FRED")
```

La instrucción USR es igual que cualquier función en BASIC, como por ejemplo la SIN() o la del COS(); también es correcta la instrucción A = USR(5), sólo que la instrucción PRINT escribirá algo en la pantalla después de que el programa en lenguaje máquina señalado por la instrucción DEFUSR haya sido ejecutado.

Hemos visto que se necesita un parámetro para utilizar la instrucción USR, pero, ¿para qué sirve eso exactamente? Continuemos, y veamos que puede ser muy útil usar un parámetro.

Los párametros y su utilidad

Sencillamente, el parámetro nos permite pasar valores del BASIC a nuestros programas en código máquina con el mínimo esfuerzo y la máxima agilidad. Seguro que ya has usado algún parámetro en tus programas; por ejemplo, cuando llamas del BASIC la función SIN() (función seno), el número entre paréntesis es un parámetro que se pasa al intérprete BASIC para calcular el seno de ese número. Al llamar a la USR, ésta nos permite coger el parámetro para que pueda ser usado por el programa en código máquina.

Al llamar a la USR, una cierta zona de trabajo del sistema tiene una de las dos informaciones sobre el parámetro que se pasó en la llamada a la USR.

a) La primera información es el valor real del parámetro, representado de la manera más apropiada.
b) La información de en qué parte dentro de la memoria del ordenador podemos encontrar el parámetro.

A esta área de memoria se le llama ZONA DE PARAMETRO (*Parameter block*, en inglés), denominación que explica muy bien lo que hace: es el área de memoria; es decir, un bloque de memoria que pasa parámetros entre el BASIC y el código máquina. Pasemos ahora a ver la configuración de la zona de parámetro. Dicha zona está en dos áreas diferentes dentro de la zona de trabajo del sistema; de esas áreas, una ocupa 1 byte y la otra 8.

La posición 63075 de la zona de trabajo del sistema se usa para indicar a tu programa en código máquina qué tipo de parámetro fue el que se pasó con la función USR. Tan pronto como escribas una instrucción USR, el intérprete de BASIC hallará el tipo de parámetro que va a ser pasado y asignará esa posición de memoria de acuerdo con dicho tipo. El resto de la zona de parámetro, es decir, los 8 bytes entre la dirección 63478 y la 63485, inclusive, se asignan entonces antes de que el intérprete dé el control a tu programa en código máquina.

Obviamente los valores que van a este bloque dependerán del tipo de parámetro llevado al bloque por la instrucción USR y el valor de ese parámetro. Los únicos tipos de parámetro que usaremos en este libro son los enteros y las cadenas alfanuméricas; los números reales quedan fuera del alcance de este libro por razones obvias.

Los parámetros enteros

Estos parámetros engloban a los números enteros, a las variables definidas como enteras; es decir, al usar "%" o la instrucción DEFINT.

Ejemplos de parámetros enteros son el 345 y el A%. En cualquiera de los casos, a la posición 63075 se le asigna el número 2, indicando que el parámetro que se pasó es un entero.

¿Qué pasa entonces con el valor del entero?

Pues bien, éste se almacena en los bytes &HF7F8 y &H77F9, que están en la parte de los 8 bytes de la zona de parámetro. Recuerda que dichas direcciones están en hexadecimal. Los enteros MSX tienen signo, por ello se almacenan en complemento a dos en estas dos posiciones de memoria.

Pasemos ahora a explicar cómo se almacenan los enteros en esas dos posiciones de memoria; tomemos, por ejemplo, el entero 1000. Anteriormente ya vimos cómo se almacenaban en los registros de la UCP los números de 16 bits; bien, pues se aplica la misma teoría para almacenarlos en posiciones de memoria. ¿Recuerdas que teníamos byte bajo y byte alto? Bueno, pues algo parecido tenemos aquí. Si representamos el 1000 para que pueda ser almacenado en el registro BC de la UCP, sería el registro B el que contendría el byte alto del número, en este caso el 3, y el registro C contendría el byte bajo; es decir, el valor 232.

Pero ¿cuáles son los números almacenados en los dos bytes &HF7F8 y &HF7F9? El byte bajo del número se almacena en la posición &HF7F8 y el byte alto en la &HF7F9. Es importante que recuerdes esto: en cualquier sitio donde se almacene en dos posiciones de memoria un número de 16 bits, el byte alto del número siempre se almacena en la posición de memoria numerada mayor, y el byte bajo del número se almacena en la posición cuyo número es menor de entre los dos. Así, en este caso concreto, el 3 se almacenaría en la posición &HF7F9 y el 232 en la &HF7F8.

Cuando la UCP transfiere números entre ella y la memoria, suele tomar en cuenta lo que te acabo de explicar, por lo que la mayoría de veces no tendrás que preocuparte de asignar cada byte en la posición indicada. En este caso, el intérprete BASIC se cuida de poner el número de 16 bits en las posiciones de memoria correspondientes.

Este método de almacenar números de 16 bits en la memoria del ordenador con el byte más bajo primero puede parecerte un tanto raro, pero no te preocupes, ya que pronto te acostumbrarás a este método.

Los parámetros alfanuméricos

Si una cadena alfanumérica, ya sea una constante o una variable, como "FRED" o A$ se pasa como parámetro a un programa en código máquina con la instrucción USR, la posición 63075 tendrá el valor 3, que indica que se ha pasado un parámetro alfanumérico.

Un parámetro alfanumérico puede constar de hasta 255 caracteres. Sin embar-

go, si la zona de parámetro usada para almacenar los parámetros sólo ocupa 8 bytes, ¿cómo se explica que pueda contener una cadena tan larga? La respuesta es bien lógica: no puede.

El contenido de la zona de parámetro actúa como un puntero de zona de memoria que nos indica dónde está la secuencia de bytes que representan a la cadena de caracteres. Esta zona de memoria que nos indica dónde están situadas las cadenas se llama DESCRIPTOR DE CADENA (*String descriptor,* en inglés).

El primer byte de esta descripción de la cadena lo indica el contenido de las posiciones &HF7F8 y &HF7F9. La posición &HF7F8 tiene el byte bajo de la dirección de la descripción de la cadena y la posición &HF7F9 tiene el byte alto de la dirección de la descripción.

Para hallar el número de 16 bits representado por 2 bytes usa la fórmula siguiente:

VALOR = VALOR DEL SEGUNDO BYTE + (256 * VALOR DEL PRIMER BYTE)

Obviamente, en este ejemplo particular la primera posición sería la &HF7F8, y la segunda sería la &HF7F9.

La descripción de la cadena sólo ocupa 3 bytes; el primer byte, señalado por la zona del parámetro, contiene el número de caracteres de la cadena; los dos siguientes bytes tienen la dirección de memoria del primer carácter de la cadena; una vez más, el byte bajo de la dirección se almacena el primero. Así, el segundo byte de la descripción de la cadena tiene el byte bajo de la dirección de la cadena, y el tercer byte tiene el byte alto de dicha dirección.

Puede que todo este proceso te parezca un poco pesado, pero la verdad es que en la práctica funciona muy bien.

Hemos visto cómo pasar parámetros enteros y cadenas alfanuméricas a nuestras rutinas en código máquina. Pasar números reales a código máquina es muy fácil, pero no va a ser explicado, ya que no creo que tú, como principiante, hagas uso de dichos números en rutinas en lenguaje máquina hasta que no tengas más experiencia.

Una vez que hayas pasado un valor a la zona de parámetro y esté siendo ejecutado nuestro programa en código máquina, puede que quieras devolver dicho valor desde los registros de la UCP al BASIC.

Este proceso de transferencia de un número desde el código máquina al BASIC se llama DEVOLVER un valor al BASIC.

Normalmente, si no escribes tus rutinas en código máquina de forma que indiques al intérprete de BASIC que quieres devolver un valor al BASIC después de ejecutar tu programa en código máquina, entonces el valor que te devolverá será el parámetro que originalmente pasaste a la rutina al utilizar la instrucción USR.

De esta manera, y teniendo en cuenta que este programa en código máquina no devuelve un parámetro, la instrucción

```
PRINT USR(7)
```

escribirá el valor 7 en la pantalla. La instrucción LET A = USR(7) nos daría que a la variable A en BASIC se le ha asignado el valor 7.

Esto no es demasiado útil; podría serlo mucho más si pudiéramos devolver el resultado de un programa en código máquina al BASIC, donde podría utilizarse.

Cómo devolver valores al BASIC

Devolver valores al BASIC es un proceso muy fácil de realizar. Como bien imaginas, es igual que el que hemos visto antes de pasar un parámetro a una rutina en código máquina, pero al contrario. Tu programa en código máquina deberá reasignar a la zona de parámetro para que, cuando el intérprete BASIC coja el control después de ejecutar tu programa en código máquina, sepa que quieres llevar un valor de vuelta al BASIC.

Puede que esto te parezca un proceso muy difícil, pero te aseguro que es bastante sencillo, ya que la ROM del MSX hace todo el trabajo difícil.

Cómo devolver números enteros

Vamos a suponer que tenemos un entero en el par de registros BC y lo queremos pasar de nuevo al BASIC; este número podría ser, por ejemplo, el resultado de un programa en código máquina. Lo primero que tendremos que hacer será asignar el valor 2 en la posición 63075. Así se indicará al ordenador que quieres devolver un entero al BASIC. Después pasamos el contenido del registro BC a las posiciones de memoria &HF7F8 y a la &HF7F9; de este modo dichas posiciones tendrán el número. En próximos capítulos veremos las instrucciones que se necesitan para hacer lo que te expliqué anteriormente.

Recuerda que el número que hemos puesto en esas dos posiciones de memoria lo tratará la ROM del MSX cuando volvamos al BASIC, como un número en complemento a dos; así, cualquier valor que pongamos, si es superior a 32767, será considerado un número negativo.

Cómo devolver cadenas alfanuméricas

Devolver cadenas alfanuméricas al BASIC es un poco más complicado que las operaciones anteriores, aunque sigue siendo muy fácil de realizar. Lo primero que tienes que hacer es asignar el valor 3 a la posición 63075; así se indica a la ROM del MSX que queremos devolver una cadena alfanumérica al BASIC. El paso siguiente es encontrar la dirección de la secuencia de bytes que representa tu cadena de caracteres.

La dirección de la cadena se define como la dirección de la posición de memoria que contiene el byte que representa el primer carácter en la cadena. También necesitamos el número de caracteres de la cadena, para lo que usamos esos 2 bits de información para fijar nuestra propia descripción de la cadena.

En el área de memoria de la descripción de la cadena que ocupa 3 bytes, pon la

longitud de la cadena (el número de caracteres de la cadena) en la primera de esas posiciones. La dirección de la cadena se pone en los otros 2 bytes, con el byte bajo de la dirección en el segundo byte de la descripción y el byte alto de la dirección en el tercer byte de dicha zona.

Las instrucciones que necesitamos para hacer lo que te acabo de explicar las veremos en los capítulos siguientes, así como algunos ejemplos sobre cómo asignar la zona de descripción del parámetro.

Lo último que tienes que hacer es asignar las posiciones &HF7F8 y la &HF7F9 para señalar al primer byte de la descripción. Como ves, aquí también se almacena el byte bajo de la dirección en la posición &HF7F8.

Para terminar la rutina en código máquina, devolverla al BASIC. El parámetro que se asignó con las técnicas que antes te expliqué, se devolverá al BASIC, en vez de al parámetro que en un principio se pasó.

Date cuenta de que es posible obtener lo que en inglés se llama un *type mismatch,* algo así como tipo erróneo; por ejemplo, si preparas la zona de parámetros para devolver una cadena alfanumérica al BASIC, y la llamada al programa en código máquina es

A% = USR(1)

Date cuenta de que no puedes asignar una cadena alfanumérica en una variable entera. Si tienes interés en saber cómo almacena el sistema BASIC las diferentes variables en memoria, tienes un apéndice al final del libro que te será de gran utilidad.

No puedes pasar a la vez más de un parámetro a código máquina, porque solamente hay una zona de parámetros; puede que esto sea una pequeña limitación si quieres pasar más de un parámetro a código máquina, en cuyo caso tendrás que usar lo que en términos informáticos se llaman "buzones". Los buzones en informática no se refieren a esas cosas color naranja que hay en las calles para echar nuestras cartas, sino que son posiciones de memoria concretas o series de posiciones desde las cuales un programa en código máquina puede coger parámetros que previamente fueron asignados ahí desde el BASIC con la instrucción POKE.

Este método no es tan "elegante" como usar la instrucción USR para pasar parámetros, pero a menudo es necesario.

De todas formas, y aunque tengamos que pasar un parámetro del programa en código máquina, cuando llamemos a la USR no tenemos que usarlo en la rutina en el lenguaje máquina. En estos casos, es decir, cuando se pasa un parámetro pero no se hace uso de él, los parámetros se llaman argumentos o parámetros mudos (*dummy,* en inglés).

Como ves, ir asimilando las instrucciones del BASIC que te permiten escribir programas en código máquina no es nada difícil; además, a estas alturas del libro ya deberías ser capaz de poner bytes en memoria, leer el contenido de posiciones de memoria y ejecutar programas en código máquina, pero todavía no sabes cómo guardar los bytes en cintas o discos. Si estudias el epígrafe siguiente, saldrás de todas las dudas que tengas sobre esto.

Las instrucciones BSAVE y BLOAD

Para guardar programas en BASIC se usan las instrucciones SAVE o CSAVE (SAVE = guardar, salvar) y para volver a cargarlas en el ordenador usamos las instrucciones LOAD o CLOAD (cargar, en español).

Las instrucciones anteriores "saben" dónde están almacenados en el ordenador los programas en BASIC, y cuando usamos dichas instrucciones lo único que éstas hacen es pasar dichos bytes al disco, o desde el disco al ordenador.

Para pasar los bytes del ordenador a la cinta, lo que necesitamos son instrucciones que indiquen las áreas de memoria que queremos guardar; dichas instrucciones son BSAVE y BLOAD (puedes pensar para recordarlas que la B que aparece en dichas instrucciones es la inicial de BYTE):

Byte LOAD   Byte SAVE

La instrucción BSAVE transporta grupos de bytes que pudieran representar, por ejemplo, un programa en lenguaje máquina a una cinta. La instrucción BSAVE completa es como sigue:

```
BSAVE   nombre, comienzo, final
```

Como ves, hay tres parámetros que ahora paso a explicar.

"Nombre"

Por "NOMBRE" entendemos el nombre del programa que va a guardarse en la cinta.

"Comienzo"

Es la dirección de la primera posición de memoria que quieres guardar en la cinta.

"Final"

Es la dirección en memoria de la última posición de memoria que quieres guardar en la cinta.

Como ya te habrás dado cuenta, lo que hace la instrucción BSAVE es guardar en un fichero de la cinta los bytes que hay comprendidos entre las direcciones "comienzo" y "final" bajo el nombre que haya asignado al parámetro "nombre". Obviamente, "nombre" es un parámetro alfanumérico, y "comienzo" y "final" son numéricos; pueden ser tanto constantes como variables.

Aún existe otro parámetro que se puede usar con la instrucción BSAVE; me refiero al parámetro *run* de tipo numérico; ejemplo:

```
BSAVE nombre,comienzo,final,run
```

Este parámetro indica la dirección de ejecución del código que va a ser guardado.

La dirección de ejecución de un fragmento de código máquina es la dirección del primer byte de la primera instrucción que va a ser ejecutada por la UCP cuando ejecutemos un programa en código máquina. Por ejemplo, la dirección de ejecución de la ROM del MSX es la dirección 0. Si no aparece el parámetro *run* en la instrucción, el ordenador cree que la dirección de ejecución del programa es la misma que la dirección dada en el parámetro ''comienzo''.

La instrucción BLOAD lleva datos de la cinta a la memoria del ordenador; la instrucción completa es:

BLOAD nombre,R,dirección de carga

Los parámetros ''R'' y *offset* (''dirección de carga'') son opcionales. Lo primero que debes saber sobre esta instrucción es que los bytes se cargan en la memoria del ordenador en la misma dirección de la cual se grabaron.

Pasemos ahora a ver estos parámetros con más detalle.

''Nombre''

Es un parámetro alfanumérico e indica a la instrucción BLOAD el nombre del fichero que quieres recuperar de la cinta; el ordenador buscará en la cinta el fichero con ese nombre hasta que lo encuentre.

''R''

Si la letra R aparece como el segundo parámetro de la instrucción, tan pronto como se cargue el programa en código máquina en cuestión, éste se ejecuta. La UCP ejecuta el programa a partir de la dirección de ejecución que se le dio cuando el programa se guardó en la cinta. Date cuenta de que, si el fichero de bytes que se carga con el parámetro ''R'' no es un programa en código máquina, lo más probable es que, cuando la UCP intente ejecutar el programa, el ordenador perderá el control y tendrás que pulsar la tecla RESET.

''Dirección de carga''

Este parámetro permite cargar un programa en una dirección distinta a la que se guardó; por ejemplo, la instrucción

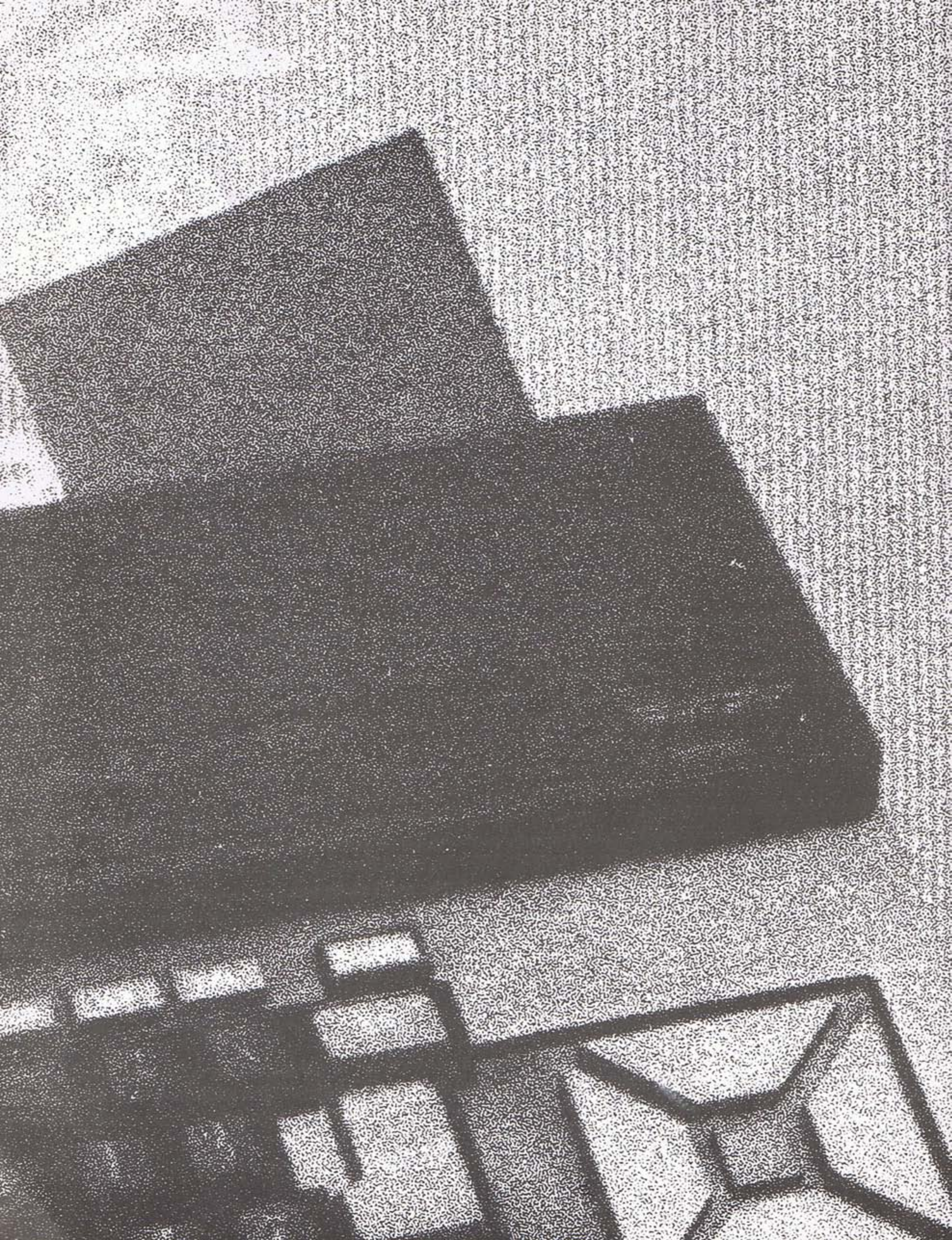
BLOAD ''FRED'',, 200

carga el fichero "FRED" en una dirección que está situada 200 posiciones más que la dirección en la que se guardó. Por ejemplo, imagínate que el parámetro de comienzo, cuando el fichero se guardó, era 60000; ahora se cargaría en la dirección 60200 si el último parámetro fuera 200. Es decir: cualquier dirección de ejecución que se guardó en el fichero queda modificada al introducir este parámetro. Si, por ejemplo, la dirección de ejecución era 60300 cuando se guardó el fichero, cuando sea cargado, la nueva dirección de ejecución será 60500.

Creo que, si has estudiado con detenimiento todo lo anterior, ya serás capaz de empezar a conocer las instrucciones que forman el código máquina del Z-80; pero antes de empezar voy a explicarte la instrucción posiblemente más importante del código máquina Z-80, que te permitirá volver al BASIC una vez ejecutados tus programas en código máquina. Dicha instrucción es RET, que es la abreviatura de RETURN. Supongo que ya la debes conocer, aunque sólo sea de oídas, ya que también existe en BASIC. Siempre que utilizamos la instrucción USR, la UCP trata nuestro fragmento en código máquina como una subrutina, a la hora de ejecutarlo. La instrucción RET es, pues, la misma que la instrucción en BASIC: RETURN, y hace que la UCP vaya al intérprete BASIC después de ejecutar nuestros programas en código máquina.

Existe una diferencia con el RETURN del BASIC, y es que, si te olvidas de algún RETURN en BASIC, provocará que salga en la pantalla un mensaje de error, o que ocurra alguna omisión del programa; en código máquina, si omites una sola orden RET perderás el control del ordenador; así que presta mucha atención.

El programa MONITOR está listado al final del libro en los apéndices. Con estas instrucciones en código máquina podrás introducir tus programas más fácilmente. Te aconsejo que lo introduzcas en el ordenador y guardes una copia, ya que los usaremos con los ejemplos.



4

Los registros en la práctica

Es obvio que no podremos realmente programar en código máquina sin introducir información en los registros Z-80. Así que en este capítulo lo que voy a hacer es explicarte las instrucciones que vas a usar para pasar números de 8 bits entre la memoria del ordenador y la UCP, así como las instrucciones que usarás para intercambiar datos entre los registros de 8 bits y la UCP. La transferencia de datos de 16 bits la veremos en el capítulo siguiente.

En informática, como en otras ciencias, existe una jerga que debes conocer antes de seguir adelante. Como ya te he dicho, estamos pasando datos de un registro a otro o a una posición de memoria; el registro del cual los datos se van a copiar se llama registro FUENTE, y el lugar en donde se copian se llama de DESTINO; se dice que estamos cargando un registro o una posición de memoria en otra. La instrucción en lenguaje ensamblador para estas transferencias es LD, abreviatura de la palabra inglesa *LoaD* (cargar).

Los diseñadores de la UCP nos dieron diferentes formas de manejar las transferencias de datos y las operaciones con datos; estas formas con las que el Z-80 usa sus registros se llaman modos de direccionamiento y se puede decir que todas las instrucciones del conjunto que tiene el Z-80 usan uno u otro modo de direccionamiento. Dichos modos te los explicaré en este capítulo.

Los diferentes modos disponibles se explicarán con referencias a las instrucciones LD, pero también se pueden aplicar con respecto a otras instrucciones. Los ejemplos de programas están en lenguaje ensamblador, por lo que será necesario traducirlo a lenguaje máquina con las tablas del final del libro. En los pri-

meros ejemplos yo mismo os listaré los códigos en hexadecimal, pero sólo en los primeros.

Lo que tienes que hacer es introducir en el ordenador los ejemplos utilizando el programa monitor del apéndice.

Veamos ahora algunas instrucciones.

Direccionamiento por registro

Se puede decir que éste es el modo de direccionamiento más sencillo que usaremos en las instrucciones que implican transferencias de datos. Lo que hace es transferir datos de un registro a otro; por ejemplo, la instrucción

```
LD A,B
```

copiará el contenido del registro B, es decir, del registro fuente, en el registro A, que es el registro de destino. Antes de seguir adelante debes saber dos cosas sobre estas transferencias: la primera es que el contenido del registro B no se ve afectado por la transferencia, y la segunda es que se pierde el contenido que hubiera en el registro A antes de la transferencia. La forma general de escribir esta instrucción es

```
LD r,r
```

donde "r" es cualquier registro de 8 bits menos el registro F. Cada una de las transferencias se codifica con un solo byte; éstos están listados en las tablas que hay al final del libro.

Por ejemplo, la instrucción en código máquina que representa la instrucción LD A,B en ensamblador es &H78. Todas las instrucciones en código máquina estarán en hexadecimal, a menos que se especifique otra cosa.

Direccionamiento inmediato

Este tipo de transferencia de datos nos permite cargar en un registro, con un programa en código máquina, valores entre 0 y 255. Los datos que se van a asignar a un determinado registro se especifican como parte de la instrucción, por lo que la instrucción ocupará 2 bytes: uno para indicar a la UCP el tipo de instrucción que va a ejecutarse, y otro para representar el valor que va a cargarse en un determinado registro. Su formato general es

```
LD r,n
```

donde ''r'' es uno de los registros de 8 bits y ''n'' es un número que se pueda representar con un byte. Por ejemplo, la instrucción

LD A,23

carga el registro A con el número decimal 23. El código para la instrucción general LD r,n tiene un nombre especial, se llama código de instrucción o código de operación; este último también se llama código *op*.

Las tablas de los códigos que representan diferentes instrucciones se llaman, como es lógico, tablas de código de instrucción o tablas de código de operación. Los bytes con datos de una instrucción determinada se llaman el BYTE DE OPERANDO.

La instrucción LD A,255 se representa en código máquina por los dos bytes 3E, FF. Igualmente, la instrucción LD B,1 se representa, en código máquina, por los dos bytes 06 y 01. El byte instrucción es el primero de los dos casos; de esta manera podemos especificar el valor real que queremos cargar en un registro.

Direccionamiento indirecto

Este caso es algo más complicado que los anteriores, pero este tipo de instrucciones son muy útiles. Los datos se transfieren entre la UCP y una posición de memoria, la dirección que está implícita en los pares de registros de 16 bits BC, DE y HL. La representación general de estas instrucciones es:

LD (rr), A
LD A, (rr)
LD (HL), n

Hay varias cosas que explicar sobre estas instrucciones:

1. El símbolo ''rr'' indica un par de registros de 16 bits.
2. Los paréntesis que contienen en los pares de registros nos dicen que estamos interesados en el contenido del par de registros. Esto ocurre en todas las instrucciones en ensamblador de la UCP Z-80.
   Siempre que aparezcan los paréntesis, recuerda que la instrucción se refiere al contenido del par de registros o a la posición de memoria en los paréntesis.
3. El par de registros HL es mucho más versátil que otros pares de registros en los que podemos cargar posiciones de memoria. Para cargar una dirección en registro HL sólo se necesita un byte, mientras que con los otros registros se necesita cargar primero en el acumulador y luego en el par de registros.

La posición de memoria cuya dirección está almacenada en el par de registros HL se dice que está direccionada por el par de registros HL. Este método de usar el par HL para cargar un byte directamente en una posición de memoria se llama:

Direccionamiento inmediato indirecto

No es necesario que te aprendas todos los nombres con los que se conoce a los modos de direccionamiento, pero sí te aconsejo que los sepas reconocer cuando te los encuentres o los leas.

Este método de usar pares de registros para "seguir la pista" a determinadas posiciones de memoria es muy útil si alguna vez necesitas escribir valores en posiciones de memoria consecutivas, todo lo que tendremos que hacer es asignar la dirección apropiada al correspondiente registro, escribir el valor deseado para esa posición, incrementar el par de registros y repetir este proceso otra vez. Las instrucciones necesarias para aumentar los pares de registros de 16 bits y para cargar direcciones directamente en ellos se explicarán más adelante en otros capítulos.

Creo que ya debes saber suficientes instrucciones para entender un programa fácil en lenguaje máquina; dicho programa hace lo mismo que la función LEN() del BASIC; con esta rutina utilizarás en la práctica las diferentes instrucciones que hemos ido viendo y aprenderás a utilizar la zona de parámetro. Si todavía no sabes bien cómo la función USR pasa parámetros a programas en código máquina, vuelve a leer el capítulo anterior con más atención.

Al final del listado de este programa te explicaré con más detenimiento lo que hace. Cuando grabes el programa monitor, empieza tecleando el código máquina en la dirección 58000; esto lo puedes hacer seleccionando la Opción 1 y tecleando 58000 como respuesta a la primera pregunta. Cuando teclees los bytes no te olvides de rellenar con ceros los bytes que contengan cifras de un solo dígito; por ejemplo:

EJEMPLO 1

Si tienes que cargar la cifra hexadecimal A, escribe 0A, la cifra 1 como 01, etc.

```
LD   H,&HF7       26   F7
LD   L,&HF8       2E   F8
LD   A,(HL)       7E
LD   C,A          4F
LD   L,&HF9       2E   F9
LD   A,(HL)       7E
LD   B,A          47
LD   A,(BC)       0A
LD   L,&HF8       2E   F8
LD   (HL),A       77
LD   L,&HF9       2E   F9
LD   (HL),0       36   00
LD   A,2          3E   02
LD   H,&HF6       26   F6
LD   L,&H63       2E   63
LD   (HL),A       77
RET               C9
```

Teclea los bytes de izquierda a derecha y hacia abajo; en este caso deberás empezar por los bytes:

26, F7, 2E, F8, 7E... etc.

Pasemos ahora a estudiar el programa con más detenimiento y ver qué hace exactamente cada instrucción.

Las dos líneas primeras cargan el par de registros HL con la dirección en la zona de parámetro que tiene el byte bajo de la dirección de zona de descripción de cadenas.

Fíjate en que el registro L tiene el byte bajo de la dirección y el H el byte alto.

La instrucción siguiente carga en el registro A el byte bajo de la dirección de descripción de cadenas; después pasamos este byte al registro C con la instrucción 4. La línea siguiente carga el registro L con el byte bajo de la dirección en la zona de parámetro que tiene el byte alto de la dirección de zona de descripción de cadenas. El byte alto de la dirección dentro del parámetro es el mismo, por lo que no tenemos que cambiarlo.

La instrucción 6 lleva el byte alto de la dirección de descripción de cadenas al registro A y después lo transfiere al registro B.

Lo que hemos hecho hasta ahora es traer la dirección de zona de descripción de cadenas que estaba en la parte de la zona del parámetro y llevarla al par de registros BC. Esta dirección, debido a que es la primera dirección de zona de descripción de cadenas, tiene la longitud de la cadena que la función USR llevó al programa en código máquina; así la siguiente instrucción usa el registro BC para recuperar la longitud de la cadena alfanumérica del registro A.

Todo lo que tenemos que hacer es fijar el parámetro para que devuelva el valor anterior al BASIC. La longitud de la cadena, como es lógico, será un número entero y deberá estar en una de estas dos posiciones: &HF7F8 o &HF7F9, de modo que el byte bajo vuelva a la posición &HF7F8. Puedes volver al capítulo 3 para más información sobre esto último.

Lo que te acabo de explicar se hace con las instrucciones que empiezan con LD L,&HF8; ésta, en concreto, asigna el par de registros HL para que señale la posición &HF7F8. La instrucción siguiente copia en una posición de memoria direccionada por el par HL el contenido del registro A. Así que ya hemos puesto el byte bajo del entero en "su" sitio. Debido a que el valor mayor que se puede recobrar nunca será superior a 255, podemos fijar, con las siguientes dos instrucciones, el byte superior a 0:

```
LD L,&HF9
LD (HL),0
```

Lo único que queda ya por hacer es asignar a la posición 63075 (&HF66) el valor 2 para indicar que queremos pasar un número entero al lenguaje BASIC; para eso usamos las tres órdenes siguientes del programa.

Como es lógico, y para terminar, usamos la instrucción RET.

Una vez que hayan sido cargados los códigos en hexadecimal, podremos ejecutar nuestro programa en lenguaje máquina; asegúrate de que has guardado una copia en la cinta.

Cómo ejecutar un programa

Para parar el programa monitor en BASIC pulsa CTRL-STOP; después, en modo directo, selecciona las siguientes instrucciones:

```
DEFUSR0 = 58000
PRINT USR0 ("TEST")
```

Si has seleccionado correctamente lo anterior, te saldrá un 4 en la pantalla, pero si lo has hecho mal y siempre que no hayas perdido el control del ordenador, vuelve a ejecutar el programa monitor y con la opción LIST repasa los bytes que hay a continuación de 58000; compruébalos con detenimiento con el listado anterior.

Si quieres, puedes incorporar el programa en código máquina anterior a un programa en BASIC, poniendo en una instrucción DATA los bytes que forman el programa y usando la instrucción READ para leerlos de la instrucción DATA y llevarlos a la RAM con instrucciones POKE.

La instrucción DEFUSR puede ser una línea del programa. Una instrucción del tipo:

```
a = USR ("test")
```

asignará a la variable de la izquierda del signo "=" cualquier valor que dé la rutina en código máquina. Pasemos ahora a estudiar otros modos de direccionamiento.

Direccionamiento extendido

Las instrucciones que usan este modo de direccionamiento tienen el siguiente formato:

LD A,(nn) y LD (nn),A

en donde *nn* es un número de dos bytes que representa una dirección.

Por ejemplo: instrucciones que usan este tipo de direccionamiento podrían ser:

```
LD a,(58010)
LD (&HF7F8),A
```

También hay instrucciones que pueden manejar registros de 16 bits, pero éstas se discutirán en otro capítulo.

El código para las instrucciones en el modo de direccionamiento anterior son:

LD A,(nn) 3A Byte bajo de la dirección. Byte alto de la dirección
LD (nn),A 32 Byte bajo de la dirección. Byte alto de la dirección

Como ves, cada instrucción necesita dos bytes de datos y un byte de código de operación. Podemos deducir de lo anterior que la dirección con la que va a trabajar la UCP aparece con el byte bajo en primer lugar.

Por ejemplo:

LD A,(&HF6F8) se codifica 3A, F8, F6

Algunas zonas del programa del ejemplo anterior se podrían haber simplificado; las instrucciones que se podrían haber resumido son:

```
LD H,&HF7
LD L,&HF8
LD A,(HL)
```

que pueden ser escritas como una sola instrucción:

```
LD A,(&HF7F8)
```

De este modo el acumulador tendría la información contenida en esa dirección. Cuando se utilice la instrucción

```
LD (&HF7F8),A
```

se copiará el contenido en ese momento del acumulador en la posición especificada de memoria. Incluso en el caso de que la dirección a la que se va a acceder sea un número que pueda representarse con un solo byte, es necesario que se especifique la dirección en la orden con dos bytes. De esta manera el byte alto de este tipo de posición de memoria siempre será 0. Por ejemplo:

LD A,(&FE) se codifica 3A FE 00

Es importante que recuerdes esto: debido a que la UCP Z-80 espera que almacenes la dirección en dos bytes, como te acabo de indicar, si olvidas uno de los dos bytes de la dirección, el Z-80 tomará el primer byte de la instrucción siguiente como parte de la dirección.

Etiquetas en lenguaje máquina

Normalmente resulta muy pesado, al escribir programas en ensamblador, tener que recordar posiciones concretas de memoria que hayamos usado para almacenar valores; por eso, y para que nos sea más fácil, se suele dar un nombre determinado a esas posiciones de memoria. Dichos nombres se llaman ETIQUETAS. Por ejemplo, la posición de memoria &H101F se podría llamar FRED. Y la instrucción que tuviera dicha posición de memoria quedaría

LD A,(FRED)

que equivaldría a la instrucción en ensamblador

LD A,(&H101F)

El uso de nombre, especialmente cuando éstos tienen connotaciones relativas al uso de esa posición de memoria, hace los listados en ensamblador mucho más legibles. Todo lo que tienes que hacer es recordar poner las direcciones en el programa cuando lo estés ensamblando a mano.

Si consigues un programa ensamblador para tu MSX, dicho programa te resolverá las etiquetas automáticamente.

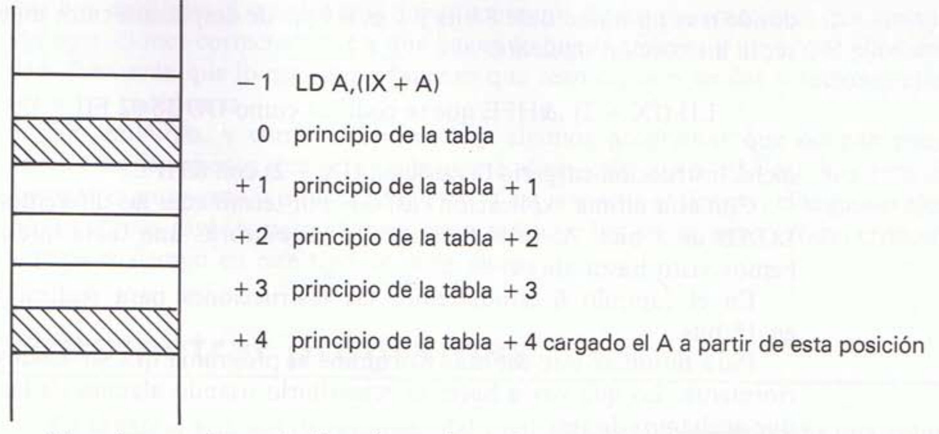
Direccionamiento indexado

Este tipo de direccionamiento usa los registros IX e IY, registros que, como sabes, no hemos usado todavía. Como en todos los métodos que hemos visto, éste también lo usan otros tipos de instrucciones, no sólo las LOAD.

Las instrucciones de direccionamiento indexado usadas para transferir números de 8 bits tienen el formato siguiente:

LD r, (IX + d)
LD r, (IY + d)
LD (IX + d), r
LD (IY + d), r

donde *d* es un número de 8 bits en complemento a dos. A este número se le llama DESPLAZAMIENTO. Las *r* que aparecen son registros de 8 bits. El desplazamiento es un número entre — 128 y + 127. Para saber lo que hace, fíjate en el siguiente dibujo que representa una zona de memoria. Supón de que la dirección de la posición de memoria que hemos llamado "Principio de la tabla" esté ya en el registro IX.



Estas instrucciones de direccionamiento indexado son muy útiles cuando tengas que acceder a datos almacenados en tablas de la memoria. En ese caso, el código que representa la instrucción tiene dos bytes; también hay otro byte en la instrucción que representa el byte de desplazamiento.

Así, la instrucción

LD A,(IX + 4)   se codifica DD 7E 04

Los bytes DD y 7E son el código de operación para esa instrucción y el 4 es el byte de desplazamiento.

Para cargar el acumulador con la posición inicial de memoria de la tabla — 1, tendremos que asignar al byte de desplazamiento el valor — 1.

Todo lo que tienes que recordar es fijar el byte de desplazamiento en complemento a dos para las posiciones negativas, así quedará que

LD A,(IX + &HFF)   se codifica como DD 7E FF

La instrucción anterior cargará el registro A con comienzo de la tabla — 1. Las tablas del código *op* para las instrucciones de direccionamiento indexado las encontrarás al final del libro.

Recordarás que te dije antes que era posible cargar un byte en una posición de memoria cuya dirección estuviera en el par de registros HL; pues bien, algo parecido podemos hacer con los registros indexados. El modo de direccionamiento se suele llamar inmediato indexado.

Direccionamiento inmediato indexado

Simplemente carga una posición de memoria cuya dirección venga dada en los registros IX e IY y su desplazamiento con un número; los formatos generales son

LD (IY + d), n
LD (IX + d), n

donde *n* es un número de 8 bits y *d* es el byte de desplazamiento; un ejemplo puede ser la instrucción siguiente:

LD (IX + 2) ,&HFE que se codifica como DD 36 02 FE

dicha instrucción cargaría la posición (IX + 2) con &HFE.

Con esta última explicación casi doy por terminadas las diferentes instrucciones LOAD de 8 bits. Al final del capítulo encontrarás una tabla que resume lo que hemos visto hasta ahora.

En el capítulo 6 estudiaremos las instrucciones para realizar transferencias en 16 bits.

Para terminar este capítulo volvamos al programa que sirvió de ejemplo anteriormente. Lo que voy a hacer es reescribirlo usando algunas de las instrucciones que acabamos de ver.

EJEMPLO 2

```
LD   A,(&HF7F8)        3A  F8  F7
LD   C,A               4F
LD   A,(&HF7F9)        3A  F9  F7
LD   B,A               47
LD   A,(BC)            0A
LD   L,&HF8            2E  F8
LD   H,&HF7            26  F7
LD   (HL),A            77
LD   L,&HF9            2E  F9
LD   (HL),0            36  00
LD   A,2               3E  02
LD   (&HF663),A        32  63  F6
RET                    C9
```

En comparación con el primer ejemplo, el segundo utiliza menos bytes para llevar a cabo el mismo programa. Hay aún una última versión que usa registros IX; en este ejemplo he tenido que introducir una instrucción que todavía no he explicado con detalle; me refiero a la instrucción que uso para introducir una dirección en el registro IX, que te explicaré en el capítulo 6.

EJEMPLO 3

```
LD    IX,&HF7F8        DD  21  F8  F7
LD    A,(IX+0)         DD  7E  00
LD    C,A              4F
LD    A,(IX+1)         DD  7E  01
LD    B,A              47
LD    A,(BC)           0A
LD    (IX+0),A         DD  77  00
LD    (IX+1),0         DD  36  01  00
LD    A,2              3E  02
LD    (&HF663),A       32  63  F6
RET                    C9
```

Como ya habrás deducido, de los tres ejemplos anteriores se puede decir que no hay una sola y válida forma de hacer un programa en código máquina, sino

que hay múltiples. Lo único que debes conseguir de tus programas es que ejecuten sus operaciones correctamente y que hagan lo que tú quieras y no lo que ellos decidan. Recuerda que lo más importante es que sean capaces de dar soluciones eficaces a tus problemas.

Sin embargo, y como es lógico, hay algunos programas que ocupan menos bytes y tardan menos tiempo en ejecutarse al ser más cortos que otros, y esto casi siempre es una ventaja a tener en cuenta. Con respecto al tiempo, el segundo ejemplo es el más rápido, pero sólo un poco más que los otros, pero ¿cómo cronometramos el tiempo en este tipo de programas?

Cómo cronometrar programas

En la tabla, a modo de resumen, del final del capítulo encontrarás una columna llamada "TIEMPO"; por ahora imagínate que dichos tiempos no son absolutos, sino que están medidos en relación con los demás tiempos, es decir, son tiempos relativos. Para aquellos de vosotros que estéis interesados en los tiempos absolutos o reales en microsegundos podéis consultar el apéndice 4.

Para hallar el tiempo de ejecución de una rutina pequeña como los ejemplos que hemos visto, te sugiero que utilices el siguiente método:

Pon en un bucle la función USR que llamará a la rutina, por ejemplo:

```
TIME = 0:FOR I = 0 TO 1000:L = USR("test"):NEXT:PRINT TIME
```

El número que salga en la pantalla te indicará el tiempo de 1.000 ejecuciones del bucle; si quitas la instrucción USR, te dará el tiempo del bucle y con una sencilla resta puedes hallar el tiempo que tardó en ejecutarse 1.000 veces la función USR.

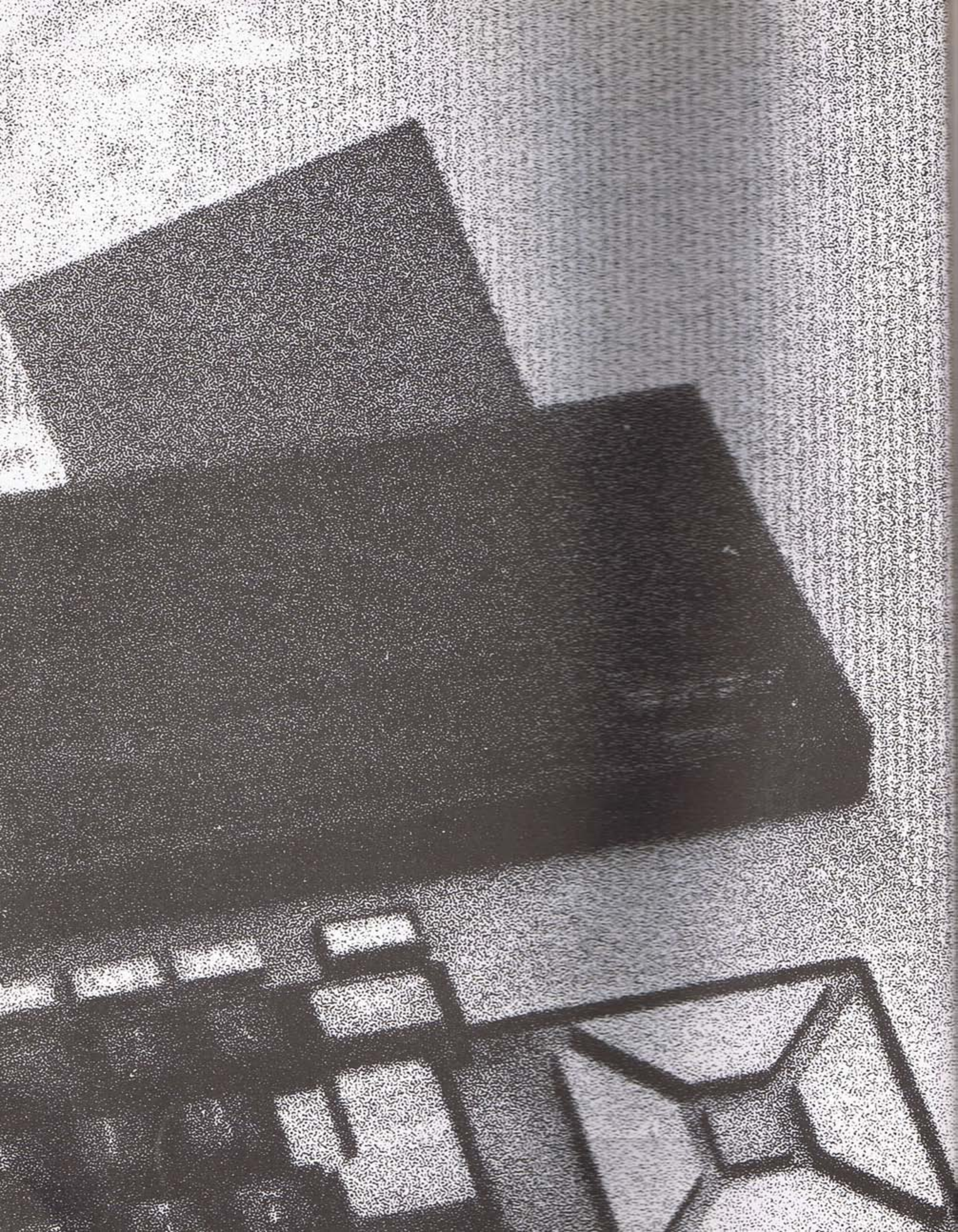
En el capítulo siguiente se estudiarán las instrucciones en 8 bits que se encargan de hacer operaciones lógicas o matemáticas; incluso veremos un par de ejemplos que interactúan con la PSG y el VDP.

Mnemónico	*Bytes*	*Tiempo*	*Efectos en los* flags *C*	*Z*	*P/V*	*S*	*N*	*H*
LD Registro, Registro	1	4	—	—	—	—	—	—
LD Registro, Número	2	7	—	—	—	—	—	—
LD A, (Dirección)	3	13	—	—	—	—	—	—
LD (Dirección), A	3	13	—	—	—	—	—	—
LD Registro, (HL)	1	7	—	—	—	—	—	—
LD A, (BC)	1	7	—	—	—	—	—	—
LD A, (DE)	1	7	—	—	—	—	—	—
LD (HL), Registro	1	7	—	—	—	—	—	—
LD (BC), A	1	7	—	—	—	—	—	—
LD (DE), A	1	7	—	—	—	—	—	—

Mnemónico	Bytes	Tiempo	C	Z	P/V	S	N	H
			Efectos en los flags					
LD Registro, (IX + d)	3	19	—	—	—	—	—	—
LD Registro, (IY + d)	3	19	—	—	—	—	—	—
LD (IX + d), Registro	3	19	—	—	—	—	—	—
LD (IY + d), Registro	3	19	—	—	—	—	—	—
LD (HL), Número	2	10	—	—	—	—	—	—
LD (IX + d), número	4	19	—	—	—	—	—	—
LD (IY + d), número	4	19	—	—	—	—	—	—

Notación para los flags:

- * indica que una operación ha afectado ese *flag*.
- 0 indica que el *flag* está a 0.
- 1 indica que el *flag* está a 1.
- — indica que no ha sido afectado ese *flag*.



5 Cálculos con 8 bits

En el capítulo 4 vimos lo fácil que es transferir datos entre los registros de la UCP, y la UCP y la memoria del MSX. También vimos cómo usando instrucciones muy fáciles podíamos escribir pequeños programas en código máquina que nos daban la longitud de la cadena como parámetro. Sin embargo, al trabajar con programas en lenguaje máquina más complicados seguimos usando sólo instrucciones que transfieren datos; estas instrucciones digamos que se nos quedan ya "cortas" para nuestros objetivos. Por eso en este capítulo veremos las instrucciones que vamos a usar para realizar cálculos y operaciones aritméticas con 8 bits, así como también operaciones lógicas. Sin embargo, antes de explicártelas, sería conveniente que entendieras bien la función de los registros llamados *flags* (INDICADORES), ya que son muy importantes a la hora de programar con instrucciones aritméticas.

El registro *flag* o indicador

El registro *flag* sirve, como ya dije en otro capítulo, para indicar ciertos sucesos o "cosas" que vayan ocurriendo; estas "cosas" a las que me refiero están relacionadas con operaciones aritméticas y lógicas de la UCP, sin importar que éstas sean de 8 ó 16 bits. Con el registro indicador nunca se trabaja como si fuera un byte, sino que se usa como una serie de diferentes bits, donde cada bit representa un indicador distinto. Esto último no es rigurosamente cierto, ya que, aunque hay

8 bits en el registro, sólo hay seis *flags;* supongo yo que al diseñador se le agotaron las ideas para usar los otros 2 bits...

Como ya dije, si se fija un bit a 1, el *flag* que representa ese bit está en SET, es decir, activado. Si por el contrario un bit está a 0, se dice que el *flag* está en RESET o CLEAR, es decir, a 0.

Después que ha sido ejecutada por la UCP una instrucción, el *flag* ''afectado'' se activará automáticamente. Los indicadores no son sensibles a todas las instrucciones; por ejemplo, las instrucciones que transfieren datos de 8 bits, que vimos en el capítulo anterior, no afectan para nada a los *flags.*

Para qué sirven los *flags* o indicadores

El esquema siguiente muestra cómo están organizados los *flags* en el registro:

7	6	5	4	3	2	1	0
S	Z	X	H	X	P/V	N	C

El significado de cada bit será explicado más adelante; antes hay dos cosas que debes tener en cuenta:

1. A los bits 3 y 5 del byte se les da el nombre ''X'' y no son importantes para las operaciones de la UCP. La letra ''X'' se usa, en estos casos, para señalar que ¡SON INUTILES!
2. El bit 2 tiene dos letras; esto quiere decir que para diferentes instrucciones también representará diferentes *flags.* Este punto será explicado más adelante.

C; el flag *de acarreo*

Como recordarás, ya dije que los números con los que trabajamos van desde el 0 al 255 para 8 bits y desde el 0 al 65535 para 16. Fíjate en la siguiente suma:

```
  11111111
+        1
  --------
```

Ya sabes que, en binario, 1 + 1 = 0 y te llevas 1; al realizar la suma, quedaría:

1 00000000

Como ves, sólo con ocho ceros se representa nuestra suma de 8 bits, por lo que no obtendremos la respuesta correcta, ya que perdemos números debido a que se produce lo que se llama un DESBORDAMIENTO *(overflow).* Cuando la UCP se encuentra con este tipo de problemas, es decir, que la suma anterior nos devuelve un noveno bit, ese ''noveno'' bit es el de acarreo, es decir, de ''llevar''.

En las restas, el *flag* de acarreo se activará si la resta en concreto da un resulta-

do negativo; por ejemplo, si restas 200 — 201, el *flag* de acarreo es entonces muy importante para la aritmética del Z-80. De hecho, si te fijas, tenemos dos instrucciones que nos permiten cambiar el estado del *flag:*

SCF representada por &H37
CCF representada por &H3F

SCF es la sigla de *Set Carry Flag,* que significa Activar el *Flag* de Acarreo; al ejecutar dicha instrucción se asignará un 1 al *flag* de acarreo.

CCF responde a las iniciales de *Complement Carry Flag,* que en español sería Complementar el *Flag* de Acarreo. Al ejecutar esta instrucción, se fijará el *flag* al estado contrario en el que está; por ejemplo, si el indicador estuviera a 0 cuando se ejecute la instrucción, CCF cambiará ese valor por un 1, y viceversa.

N; el flag *de resta*

Este *flag* lo usan principalmente las instrucciones especiales aritméticas en BCD que veremos más adelante en este mismo capítulo. E indica cuando se activa, es decir, cuando se pone a 1, que la última instrucción fue una resta.

H; el flag *de medio acarreo*

Este es un indicador, digamos especial, que indica que te llevas algo del quinto bit del acumulador en vez del noveno bit. Se usa normalmente en la aritmética BCD.

P/V; el flag *de paridad/desbordamiento*

Actúa como *flag* de paridad durante las operaciones lógicas de la UCP y también como *flag* de desbordamiento en operaciones aritméticas.

Sobre las operaciones lógicas sólo diré por ahora que, si el byte tiene un número IMPAR de bits a 1 se dice que es de paridad impar y el *flag* se fijará a 0. Si, por el contrario, tiene un número PAR de bits a 1, se dice que es de paridad par y el *flag* se activará.

El *flag,* en modo desbordamiento, es muy útil cuando estés trabajando en complemento a dos, e indica que la suma de dos números positivos en complemento a dos dará un número negativo y que la suma de dos números negativos en complemento a dos da un número positivo.

Ninguno de estos resultados son válidos, y así sabremos que se ha producido un desbordamiento. Si no prestas atención a este *flag,* puede que a veces lo confundas con el *flag* de acarreo; por eso recuerda que el *flag* de acarreo se refiere al noveno bit, mientras que el de desbordamiento se refiere al octavo bit.

Z; el flag *0*

Se puede decir que éste es el indicador más útil de los disponibles: indica si el resultado de una operación ha sido 0 o no. Se refiere al registro A y, por tanto, lo que hace es comprobar si alguna operación dejó un 0 en dicho registro o no. Sin embargo, no porque cargues un 0 en el registro A se activará este indicador.

El estado del *flag* en Z es, en muchos casos, el único resultado visible de algunas operaciones.

El *flag* se fija a 0 con un resultado que no sea 0; ten cuidado con esto y no te confundas.

S; el flag *de signo*

Este *flag* se activa con las operaciones aritmético-lógicas que usan el registro A y sirve para indicar el signo del resultado de una operación. Efectivamente, es una copia del MSB del registro A, y de acuerdo con la representación en complemento a dos, se activará si el resultado es negativo y se pondrá a 0 si es positivo.

¿Cómo se utilizan los *flags*?

En BASIC, hay programas que tienen las siguientes proposiciones:

IF A = 2 THEN ...

En código máquina podemos usar el estado de varios *flags* junto con algunas instrucciones que veremos más adelante para formar construcciones parecidas. La única diferencia es que utilizaremos las instrucciones del código máquina equivalente al BASIC:

GOTO y GOSUB

Los equivalentes a esas instrucciones en código máquina son las instrucciones JP, abreviatura de *JumP* (salto, en español) y CALL (llamada, en español), que veremos en profundidad en el capítulo 8. Por ahora sólo diré que los *flags* se usan para decidir si vamos a hacer una orden JP o una CALL; por ejemplo, el equivalente en ensamblador de

IF A = 0 THEN GOTO 12000

podría ser

JP Z, 12000

Esta instrucción comprueba que el resultado de la última operación fue un 0 y si se saltó al código máquina almacenado en la posición 12000. La JP Z, 12000, es

un ejemplo concreto de las instrucciones llamadas condicionales, es decir, que sólo se ejecuta si se cumplen ciertas condiciones o el estado de diferentes *flags*.

Esta es sólo una pequeña introducción. En las tablas del final del libro hay más información; en dichas tablas se indica qué *flags* son afectados por las diferentes instrucciones.

Cómo se cuenta con 8 bits

La operación aritmética más sencilla es sumar o restar un 1 a un número. La UCP realiza esta operación con facilidad; esto es lo que vamos a ver ahora.

Cómo sumar

Para sumar un 1 al contenido de un registro usamos la instrucción:

INC r

donde *r* es un registro de 8 bits. Un ejemplo concreto de esta instrucción podría ser INC A, que sumaría 1 al contenido del registro A.

La orden

INC rr

representa una instrucción que suma un 1 al valor que haya en el registro de 16 bits. Los cálculos en 16 bits se explicarán más adelante en otros capítulos. En estas instrucciones de sumar o INCrementar se usan muchos de los modos de direccionamiento que hemos visto en las direcciones LOAD de 8 bits; por ejemplo, vamos a usar el direccionamiento indexado:

INC (IX + d)
INC (IY + d)

Determinamos la dirección de la memoria con el contenido de los registros índices y el byte de desplazamiento; después se suma un 1 al contenido de ese byte. Podemos usar el par de registros HL de forma parecida para modificar el contenido de las posiciones de memoria. Fíjate en las siguientes instrucciones:

INC (IX + d),	aumenta el contenido de la posición de memoria (IX + d), donde *d* es el byte de desplazamiento
INC (IY + d)	aumenta el contenido de la posición de memoria (IY + d)
INC (HL)	aumenta el contenido de la posición de memoria direccionado por el par de registros HL

Por ejemplo:

LD A, 254
INC A

dejaría el valor 255 en el registro A:

LD HL,58000
LD A,0
LD (58000),A
INC (HL)

asignaría un 1 en la posición 58000. Recuerda que las instrucciones cuyos paréntesis contienen pares de registros se refieren al contenido de la dirección dada por el par de registros.

EJEMPLO 4

Veamos ahora un programa completo en código máquina que use las instrucciones INC. En este caso, también usaré el procesador de video (VDP).

Deja por ahora las instrucciones que uso para escribir datos en el VDP; ya te las explicaré en los capítulos 9 y 10.

Este programa opera a través del conjunto de caracteres MSX para modificar el contenido de la RAM de video. Usaremos la instrucción INC(HL) para cambiar el carácter que va a salir en la pantalla, cada vez que usamos la instrucción USR1.

Teclea el siguiente programa monitor en tu ordenador almacenándolo en la dirección 58100; después, ejecútalo y así sabrás lo que hace:

```
LD   A,0              3E  00
LD   (58000),A        32  90  E2
LD   HL,58000         21  90  E2
INC  (HL)             34
IN   A,&H99           DB  99
LD   A,0              3E  00
OUT  &H99,A           D3  99
LD   A,64             3E  40
OUT  &H99,A           D3  99
LD   A,(58000)        3A  90  E2
OUT  &H98,A           D3  98
RET                   C9
```

Para ejecutar el programa, escribe el siguiente en BASIC después de que hayas cargado los bytes del ejemplo anterior.

```
10 SCREEN 0
20 DEFUSR0=58100
30 DEFUSR1=58105
40 F=USR0(0)
50 F=USR1(0)
60 I$=INPUT$(1):GOTO 50
```

Este tipo de programas en BASIC, que se usan para controlar la ejecución de programas en código máquina, se llaman controladores, o *driver,* en inglés.

Si el programa está funcionando y pulsas la barra espaciadora, el carácter que aparezca en la esquina superior izquierda de la pantalla cambiará cada vez que pulses dicha barra. Cada vez que digites esa tecla se ejecutará el trozo de rutina en código máquina que empieza en la dirección 58105; es decir, escribirá el contenido de la posición 58000 de la pantalla después de haber aumentado esa posición usando la instrucción INC(HL).

Con respecto al código que empieza en la dirección 58100 te diré que se usa para inicializar la posición 58000 a 0. Por esa razón, sólo se le llama una vez; todas las demás llamadas a la USR son a la dirección 58105. Las instrucciones a partir de la tercera instrucción (después de la dirección 58105) en adelante se utilizan para escribir los datos necesarios en la VRAM. Para ello se mandan datos a los registros del VDP. Esto es simplemente decirle al VDP que desea escribir datos en el primer byte de la VRAM.

En el modo de pantalla 0 un byte escrito en cualquiera de los primeros 900 bytes de la VRAM hace que aparezca un carácter en la pantalla; el byte que se va a escribir en la VRAM representa el carácter en código ASCII que queremos que salga en la pantalla.

EJEMPLO 5

Este programa también escribe en el VDP, pero esta vez lo que estamos haciendo no es escribir datos en la VRAM, sino en los registros internos del VDP, con lo cual cambiamos (en modo 0) los colores de la pantalla tanto del fondo como de la presentación.

El programa controlador que utilizamos en el ejemplo anterior nos vale para éste. También tienes que escribir el código en la dirección 58100 y volvemos a usar la orden INC(HL). Si quieres reescribir la rutina puedes cargar el contenido de la posición 58000 en el registro A, ejecuta una instrucción INC A y después vuelve a cargar en la posición 58000 el registro modificado A.

El listado para el ejemplo número 5 es el siguiente:

```
LD    A,0                3E   00
LD    (58000),A          32   90   E2
LD    HL,58000           21   90   E2
INC   (HL)               34
IN    A,&H99             DB   99
LD    A,(58000)          3A   90   E2
OUT   &H99,A             D3   99
LD    A,135              3E   87
OUT   &H99,A             D3   99
RET                      C9
```

Escribe también los bytes en hexadecimal en la posición 58100. La mayoría del programa es igual al que vimos en el ejemplo 4, pero, mientras que en ése escribíamos en la VRAM, ahora estamos escribiendo en los registros VDP. En el capítu-

lo 10 hay una explicación más detallada de cómo se hace esta operación. La zona de código entre las instrucciones

```
IN A,&H99
......
OUT &H99,A
```

escribe el valor almacenado en la posición 58000 en el registro 7 del VDP, que controla los colores que aparecen en la pantalla en modo 0.

Las instrucciones

```
LD  A,135
OUT &H99,A
```

escriben en el VDP el número del registro que nos interesa; en este caso, el 7. Dicho número se manda al VDP después de haberle sumado 128, con lo que queda explicado el número 135 que aparece en la primera instrucción.

Es posible, aunque te parezca una incongruencia, tener un 0 en un registro después de haber ralizado varias sumas; por ejemplo: si un registro tiene un valor de 255 y le sumamos un 1, asignará un 0 a los 8 bits menores, como ya hemos visto.

Cómo restar

Las instrucciones para restar son análogas a las que usábamos para sumar, es decir:

```
DEC r
DEC rr
DEC IX
DEC IY
DEC (HL)
DEC (IY + d)
DEC (IX + d)
```

Las instrucciones DEC restan un 1 al valor del registro o de la posición de memoria especificados. DEC es la abreviatura del término *DECrease* o *DECrement* (DECrementar). Si tienes un registro que posee un 0 y le restas un 1, te dará 255.

Efectos en los *flags*

Las instrucciones de 8 bits INC y DEC afectan a la mayoría de los *flags;* sin embargo, las instrucciones INC y DEC de 16 bits (pares de registros) no afectan a ningún *flag,* lo cual significa que tendremos que usar más instrucciones. Lo que nunca sabremos es si esa negligencia fue un fallo del diseñador o no.

El único *flag* importante que no se ve afectado por las instrucciones INC y DEC es el *flag* de acarreo.

Flag de signo

Este *flag* se activa si el bit número 7 de un resultado de 8 bits es 1.

Flag 0

Se activa si el resultado es 0.

Flag de desbordamiento

Se activa si una operación cambia el valor del bit número 7 de un resultado de 8 bits.

Flag de mitad de acarreo

Se activa si hay acarreo del cuarto bit en una operación de 8 bits.

Flag de negación

Se activa si la última instrucción fue DEC; es decir: una resta.

Si ejecutas las dos instrucciones siguientes, aprenderás el uso de la instrucción DEC en la práctica: en este ejemplo se pasa un valor a un programa en código máquina, se decrementa y entonces se devuelve al BASIC vía zona de parámetros. Carga el código en la posición 58000 y ejecuta las siguientes instrucciones:

```
DEFUSR = 58000
PRINT USR (n)
```

donde *n* es el valor que quieres disminuir; dicho valor tiene que ser un número de 8 bits.

EJEMPLO 6

```
LD    A,(&HF7F8)        3A  F8  F7
DEC   A                 3D
LD    (&HF7F8),A        32  F8  F7
LD    A,2               3E  02
LD    (&HF663),A        32  63  F6
RET                     C9
```

Como verás, poder sumar y restar un 1 ya es bastante útil e importante, pero no es todavía lo que hay que hacer para sumar, por ejemplo, 56 más 167. Para realizar esta operación necesitamos instrucciones que sumen 8 bits; dichas instrucciones son las que vamos a ver ahora.

Aritmética de 8 bits

En esta sección estudiaremos las operaciones aritméticas en 8 bits para sumar y restar. Todas estas operaciones usan el registro A; de hecho se puede decir que muchas de las instrucciones dan por hecho que tienen que usar el registro A y no hace falta que se mencione dicho registro en la instrucción.

Las operaciones aritméticas que la UCP sabe hacer son la suma y la resta, pero no conoce ni la multiplicación ni la división; si quieres hacer estas operaciones en código máquina, tú mismo tendrás que hacerte un programa para tal fin.

La operación más fácil y simple que veremos en esta sección es la suma con 8 bits; empecemos, pues, con ella.

Suma con 8 bits

La suma más simple con 8 bits es de esta forma:

```
ADD A,r
```

donde *r* es cualquier registro de 8 bits. Esta instrucción ordena a la UCP que sume el contenido del registro *r* al contenido del acumulador y ponga el resultado en el registro A. De ahí el origen del término acumulador; en algunos ordenadores antiguos un determinado registro se usaba para "acumular" los resultados de diferentes operaciones al igual que ahora este registro A. Por ejemplo, las siguientes instrucciones suman 20 al valor que en ese momento tenga el registro A:

```
LD   B,20
ADD A,B
```

Si te fijas en la tabla que hay al final del capítulo, te darás cuenta de que las sumas en 8 bits inciden en los valores de todos los *flags*. Fijan el *flag N* a 0 y cambian los otros de acuerdo con el resultado de cada operación.

La adición con 8 bits tiene muchos modos de direccionamiento:

```
ADD A,r
ADD A,n
ADD A,(HL)
ADD A,(IX + d)
ADD A,(IY + d)
```

El único modo de direccionamiento que cabría esperar aquí, y no está, es el siguiente:

ADD A,(nn)

Lo que debes hacer, debido a que no puedes usar esa instrucción, es simular esa operación con las siguientes instrucciones.

```
LD   HL,nn
ADD A,(HL)
```

Un punto importante que debes tener en cuenta es que el resultado de este tipo de sumas con 8 bits deben ser números en 8 bits; por ejemplo:

```
LD   A,&H80
ADD A,&H81
```

dejará en el acumulador &H01, lo cual es incorrecto, pero, sin embargo, la operación ha activado el *flag* de acarreo indicando así que el resultado necesitaba más de 8 bits para poder representarse; como ves, tenemos un *flag* que nos dice si tenemos un error causado por desbordamiento del registro A.

Veamos ahora al ejemplo de una suma con 8 bits; pasaremos dos números a un programa en lenguaje máquina; éstos se sumarán, y el resultado se devolverá al programa en BASIC.

EJEMPLO 7

Teclea el siguiente programa en la posición 58100:

```
58100  LD    A,(&HF7F8)       3A   F8   F7
       LD    (58000),A        32   90   E2
       RET                    C9
58107  LD    A,(&HF7F8)       3A   F8   F7
       LD    HL,58000         21   90   E2
       ADD   (HL)             86
       LD    (&HF7F8),A       32   F8   F7
       LD    A,2              3E   00
       LD    (&HF663),A       32   63   F6
       RET                    C9
```

Para ejecutar el programa usamos el siguiente programa en BASIC.

```
10 DEFUSR0=58100:DEFUSR1=58107
20 INPUT A%:G=USR0(A%)
30 INPUT A%:PRINT USR1(A%)
40 GOTO 20
```

Fíjate en la cantidad de programa en código máquina que se encarga de coger valores de la zona de parámetro y devolverlos otra vez después de realizarse la suma. Las primeras instrucciones aceptan el primer número de la suma y lo almacenan en la posición 58000 y vuelven al programa en BASIC. La instrucción USR0 llama a la rutina anterior, la instrucción USR1 llama a la segunda rutina, lleva el segundo sumando de la operación y ejecuta la suma utilizando la instrucción:

ADD (HL)

Después se pasa al BASIC el resultado usando la zona de parámetros. Para ver lo que este programa hace, ejecuta el programa en BASIC y responde a las preguntas que vayan apareciendo. Recuerda que los números deben estar comprendidos entre 0 y 255; si no, se te presentarán problemas debido a los desbordamientos que ocurrirán en el programa.

En el ejemplo 8 tienes otro ejemplo de programa que hace lo mismo, pero de manera diferente.

EJEMPLO 8

```
58100  LD   A,(&HF7F8)     3A  F8  F7
       LD   (58000),A      32  90  E2
       RET                 C9
58107  LD   A,(&HF7F8)     3A  F8  F7
       LD   B,A            47
       LD   A,(58000)      3A  90  E2
       ADD  A,B            80
       LD   (&HF7F8),A     32  F8  F7
       LD   A,2            3E  02
       LD   (&HF663),A     32  63  F6
       RET                 C9
```

Si quieres hacer un programa que tome el número del byte bajo de la zona de parámetro, que después le sume una constante y que devuelva el resultado al BASIC, usa la instrucción

ADD A,n

donde *n* es un número de 8 bits.

Si has ejecutado cualquiera de las rutinas anteriores, habrás realizado algo parecido a:

? 123 como primer sumando, y
? 245 como segundo
112

El resultado debería ser 368 y no 112; lo más lógico es que pienses que nuestra rutina USR tiene un fallo, pero no ha sido así. Lo que ha ocurrido es que ha habido un error de desbordamiento:

112 = 368 — 256

¿Qué podremos hacer para evitar este error? La respuesta es usar la instrucción ADC; dicha sigla responde a las iniciales inglesas *ADd with Carry,* que quiere decir Suma con Acarreo.

Suma con acarreo

Estas instrucciones funcionan igual que las del tipo ADD; tienen los mismos modos de direccionamiento y funcionan con registros y números de 8 bits.

Pero, a diferencia de las instrucciones ADD, éstas nos permiten hacer operaciones con números mayores que 255.

La diferencia entre las instrucciones ADC y las ADD es que el valor del *flag* de acarreo, ya sea un 1 o un 0, se suma al resultado que haya en el registro A. Como ejemplo, vamos a realizar una suma cuyo resultado sea un número de 2 bytes: sea la suma 100 más 200; el resultado lo vamos a almacenar en las posiciones de memoria 58000 y 58001. Como ya veníamos haciendo, el byte bajo se almacena en la dirección 58000 y el byte alto en la 58001. Además tomamos los dos números que vamos a sumar como números de 2 bytes:

EJEMPLO 9

```
LD    A,200           ;byte bajo del primer numero en A
LD    B,100           ;byte bajo del segundo numero en B
ADD   A,B             ;suma los dos numeros
LD    (58000),A       ;almacena el byte bajo
LD    A,0             ;byte alto del primer numero en A
LD    B,0             ;byte alto del segundo numero en B
ADC   A,B             ;suma los dos registros e incluye el
                      ;acarreo
LD    (58001),A       ;almacena el byte alto
RET
```

Si quieres introducir este ejemplo en tu ordenador, encontrarás el código *op* para cada instrucción en las tablas que hay al final del libro. No olvides poner todas las direcciones del programa con los bytes bajos primero. Cárgalo en la dirección 58100 con la instrucción DEFUSR = 58100 y la PRINT USR(0) para ejecutarlo; para saber el resultado, teclea

PRINT PEEK(58000) + 256 * PEEK(58001)

Pasemos ahora al ejemplo 10, algo más complicado que este último que hemos visto. Recuerda, a la hora de realizar estas operaciones, que cualquier entero en MSX que tenga su MSB a 1, el ordenador lo tomará como un número negativo en complemento a dos; por esto el número máximo positivo que puede tomarse como tal y que, por tanto, será devuelto como un entero por el BASIC del MSX es 32767.

Una vez que estés en el BASIC, si quieres pasar un número en complemento a dos a uno positivo, recuerda que esta representación de enteros de 16 bits como números en complemento a dos es una función del ordenador MSX y no del pro-

cesador Z-80. Por ejemplo, podemos tener un programa en código máquina que sume dos números enteros de 32 bits y que devuelva los valores como números positivos asignando los bytes resultantes directamente a la memoria y no a través de la zona de parámetros.

EJEMPLO 10

```
58100  LD   A,(&HF7F8)
       LD   (58000),A
       LD   A,(&HF7F9)
       LD   (58001),A
       RET
58113  LD   A,(&HF7F8)
       LD   HL,58000
       ADD  A,(HL)
       LD   (&HF7F8),A
       LD   HL,58001
       LD   A,(&HF7F9)
       ADC  A,(HL)
       LD   (&HF7F9),A
       LD   A,02
       LD   (&HF663),A
       RET
```

Los bytes del programa los tienes al final del párrafo, pero sería más instructivo que los buscaras tú mismo y luego compararas tu lista con la que te doy. Digita el código en la dirección 58100; usa el controlador en BASIC que te facilito para ejecutar el programa.

BYTES: 3A,F8,F7,32,90,E2,3A,F9,F7,32,91,E2,C9,3A,F8,F7
21,90,E2,86,32,F8,F7,21,91,E2,3A,F9,F7,8E,32,F9,
F7,3E,02,32,63,F6,C9

```
10 DEFUSR=58000:DEFUSR1=58113
20 INPUT A%:A=USR(A%)
30 INPUT A%:PRINT USR1(A%)
40 GOTO 20
```

Esta rutina también maneja número negativos, ya que el intérprete MSX lleva los enteros en complemento a dos a la zona de parámetros cuando traspasa el número a la rutina.

Restas con 8 bits

Las instrucciones para esta operación son iguales que las de la suma; hay dos grupos de instrucciones:

SUB — restas sin acarreo
SBC — restas con acarreo

No me voy a parar en estas dos clases de instrucciones debido a su similitud con las que ya vimos para la adición; sus modos de direccionamiento también son los mismos. Si quieres hacer algunos ejercicios usa, en vez de las instrucciones para sumar, las de restar.

Aritmética BCD

Hasta ahora todas las operaciones aritméticas que hemos visto han sido en binario. En BCD, que son las siglas de *Binary Coded Decimal,* que en español significa Código Decimal en Binario, los bytes de 8 bits los descomponemos en *nibbles* de 4 bits cada uno; cada medio byte contiene un número en binario representando un dígito del 0 al 9. Lo cual significa que seis de las dieciséis combinaciones de 4 bits posibles no se usan.

Por ello, el BCD no es un método eficaz para almacenar números:

0010 0010 = 22 en BCD o 34 en decimal

igualmente:

1001 1001 = 99 en BCD

Este método de representar números ocasiona muchos problemas a la hora de sumar y restar; por ejemplo:

BCD 08 0000 1000
BCD 03 0000 0011

Si sumamos estos dos números, el resultado en binario será

0000 1011

pero éste no es el formato correcto en BCD y entonces tendremos que pasar este resultado a su formato apropiado en BCD. Esto se consigue sumando 6 al resultado, siempre que el byte bajo sea mayor que 9. Gracias a la instrucción *Decimal Adjust Accumulator* (Ajuste Decimal del Acumulador) o DAA, el acarreo del byte menor y mayor BCD nos lo indica nuestro ya viejo conocido: el *flag* H o de medio acarreo.

Aunque me haya detenido a explicarte la aritmética en BCD, es muy poco probable que la utilices en tus programas.

Cómo comparar números

Puede que a primera vista te resulte un tanto extraño este título sobre comparación de números cuando en realidad lo que estamos viendo ahora es aritmética y lógica, pero, si lo piensas, te darás cuenta de que, cuando comparas dos números, lo que en realidad estás haciendo es una resta.

El resultado de dicha resta te indica qué número de la comparación es el mayor, menor o igual. El conjunto de instrucciones en el Z-80 que se encargan de comparar números se llaman, en mnemónico, CP.

Las instrucciones CP

Las instrucciones CP comparan el contenido del registro A con cualquier otro número de 8 bits. Hay varios modos de direccionamiento, que son:

```
CP r
CP n
CP (HL)
CP (IX + d)
CP (IY + d)
```

Por ejemplo, la instrucción CP r resta el contenido del registro *r* al del registro A. Aunque en las instrucciones anteriores no se nombre el acumulador, recuerda que trabaja con el registro A.

Las instrucciones CP sí modifican los *flags*. En el caso del *flag* N se activa cuando la operación es una resta. Las instrucciones CP utilizan el *flag* P/V como desbordamiento.

Fíjate en los efectos sobre los *flags* en el ejemplo siguiente:

```
LD A,10
CP B
```

Dichas instrucciones actúan sobre los *flags* de acarreo y 0 de la siguiente manera:

Si el contenido del registro B es igual a 10,

entonces Z = 1 y C = 0.

Si el contenido de B es mayor que 10,

entonces Z = 0 y C = 1.

Si el contenido de B es menor que 10,

entonces Z = 0 y C = 0.

Las instrucciones de salto condicional utilizan dichos indicadores para controlar posibles desbordamientos en los programas. Las operaciones CP no alteran el valor del registro A. Otros ejemplos sobre cómo actúan las instrucciones CP sobre los *flags* son:

```
LD  A,23
LD  B,22
CP  B
```

El resultado será que el *flag* Z y el C se pondrán a 0.
En

```
LD  A,20
LD  B,30
CP  B
```

el *flag* Z se pondría a 0, mientras que el C se activaría. Como ya he dicho antes, el estado de los *flags* se usa para controlar el flujo de un programa. Este punto es muy importante que quede bien claro, ya que sólo analizando el estado de los indicadores podemos saber el resultado de una comparación.

Por eso mismo, debes analizar el estado de tus *flags* después de ejecutar la instrucción CP, antes de ejecutar cualquier otra instrucción que pudiera cambiar los indicadores.

Mnemónico	*Bytes*	*Tiempo*	*Efectos en los* flags C	Z	P/V	S	N	H
ADD A, registro	1	4	*	*	*	*	0	*
ADD A, número	2	7	*	*	*	*	0	*
ADD A, (HL)	1	7	*	*	*	*	0	*
ADD A, (IX + d)	3	19	*	*	*	*	0	*
ADD A, (IY + d)	3	19	*	*	*	*	0	*
ADC A, registro	1	4	*	*	*	*	0	*
ADC A, número	2	7	*	*	*	*	0	*
ADC A, (HL)	1	7	*	*	*	*	0	*
ADC A, (IX + d)	3	19	*	*	*	*	0	*
ADC A, (IY + d)	3	19	*	*	*	*	0	*
SUB registro	1	4	*	*	*	*	1	*
SUB número	2	7	*	*	*	*	1	*
SUB (HL)	1	7	*	*	*	*	1	*
SUB (IX + d)	3	19	*	*	*	*	1	*
SUB (IY + d)	3	19	*	*	*	*	1	*
SBC A, registro	1	4	*	*	*	*	1	*
SBC A, número	2	7	*	*	*	*	1	*
SBC A, (HL)	1	7	*	*	*	*	1	*
SBC A, (IX + d)	3	19	*	*	*	*	1	*
SBC A, (IY + d)	3	19	*	*	*	*	1	*
CP registro	1	4	*	*	*	*	1	*

Mnemónico	*Bytes*	*Tiempo*	*Efectos en los* flags C	Z	P/V	S	N	H
CP número	2	7	*	*	*	*	1	*
CP (HL)	1	7	*	*	*	*	1	*
CP (IX + d)	3	19	*	*	*	*	1	*
CP (IY + d)	3	19	*	*	*	*	1	*

Notación para los flags:

* indica que una operación ha afectado ese *flag*.
0 indica que el *flag* está a 0.
1 indica que el *flag* está a 1.
— indica que no ha sido afectado ese *flag*.

Operadores lógicos

Estas instrucciones son tan importantes en la programación en lenguaje máquina como lo son las sumas y las restas en la aritmética común.

Estos operadores se suelen llamar de BOOLE, debido al matemático que las estudió y definió. La diferencia entre estos operadores y los que hemos visto hasta ahora es que éstos trabajan con bits en vez de con bytes. Los operadores lógicos operan con el valor de cada bit en vez de operar con el valor numérico que el byte representa.

Hay cuatro operadores de Boole dentro de las instrucciones del Z-80:

XOR
AND
NOT
OR

Las tablas de verdad

Lo único que hacen las tablas de verdad es darte el resultado de una operación lógica con 1 ó 2 bits. Para hacer una tabla de verdad, lo único que tienes que hacer es escribir el bit o bits sobre los que actúa una determinada instrucción y luego ir anotando el resultado a continuación, formando así las tablas; no te preocupes si no te ha quedado muy claro lo que es una tabla, ya que más adelante verás ejemplos de algunas tablas.

NOT

Lo que hace la operación NOT es lo siguiente:

Operando	*Resultado*
0	1
1	0

Como verás, es una operación de complemento y a la instrucción que hace esto se le llama CPL, que complementa también el acumulador. No existen otros modos de direccionamiento para esta instrucción; un ejemplo de lo que hace esta instrucción es el siguiente:

```
CPL
INC A
```

que halla el complemento a dos de un número situado en el acumulador.

AND

Dentro del conjunto de las instrucciones del Z-80 hay once instrucciones AND diferentes, cuyos modos de direccionamiento son:

```
AND r
AND n
AND (HL)
AND (IX + d)
AND (IY + d)
```

Fíjate que aquí no se nombra el registro A en ningún modo de direccionamiento; esto se debe NO a que no lo utilicen, sino a que se da por hecho su uso.

La tabla de verdad de una operación AND para un solo bit sería:

A	*B*	*Resultado*
0	0	0
0	1	0
1	0	0
1	1	1

Lo que hace esta función es devolver un 1 si los dos bits son 1. La instrucción

```
AND B
```

ejecuta la operación AND con el registro A y el B, cuyo resultado es el siguiente:

```
        A  0000  0101
        B  0001  0000
  A AND B  0000  0000
```

Otro ejemplo puede ser:

```
        A  0000  0101
        B  0000  1111
  A AND B  0000  0101
```

Como puedes ver, la instrucción AND sirve para "encubrir" determinados bits del acumulador, un ejemplo de esto será usar la instrucción AND para asegurarnos de que un byte sólo tenga un valor de 0 a 15 inclusive. Lo que hacemos es ignorar los bytes del 4 al 7 con la instrucción AND.

```
LD   A,(58000)
LD   B,&B00001111
AND B
LD   (58000),A
```

De esta manera, no importa el valor que tengan los 4 bits altos de la posición 58000, ya que el resultado de la instrucción AND los haría todos ceros. Otro ejemplo sería:

```
LD   B,&H11101111
AND B
```

que asignaría un 0 al bit número 5 del registro A.

OR

La tabla de verdad de esta instrucción es como sigue:

A	*B*	*A OR B*
0	0	0
0	1	1
1	0	1
1	1	1

Sus modos de direccionamiento son los mismos que los de la instrucción AND. El lenguaje máquina sólo permite usar el registro A para ejecutar la instrucción OR.

Esta instrucción se suele utilizar para asignar un 1 a algunos bits del registro A, o a otros registros o a posiciones de memoria. Por ejemplo, para fijar a 1 el bit 1 del registro A se utiliza

```
LD B,2
OR B
```

o simplemente

```
OR 2
```

Como pronto veremos, existen otras instrucciones que también hacen esto.

EJEMPLO 11

El código ASCII de una letra minúscula es 35 veces mayor que su correspondiente mayúscula en el mismo código. La siguiente rutina en código máquina toma una cadena como parámetro de la función USR y lo devuelve al BASIC después de poner en minúscula la primera letra de la cadena. Haz tú lo mismo con algunos caracteres no alfabéticos y piensa qué haría el ordenador.

Esta rutina escribe en la pantalla todos los caracteres ASCII:

```
10 FOR I=0 TO 255
20 PRINT CHR$(I)
30 NEXT I
```

Los bytes que forman este programa debes teclearlos a partir de la posición 58000.

```
LD    HL,&HF7F8
LD    C,(HL)
INC   HL
LD    B,(HL)
INC   BC
LD    A,(BC)
LD    L,A
INC   BC
LD    A,(BC)
LD    H,A
LD    A,(HL)
OR    32
LD    (HL),A
LD    A,3
LD    (&HF663),A
RET
```

La mayor parte de la rutina, desde el principio hasta antes de la instrucción OR 32, sirve para conseguir el primer carácter de la cadena.

Como ves, he usado varias instrucciones de 16 bits, aunque todavía no las debería utilizar por no haberlas explicado aún; luego te diré el porqué de esto.

Lo primero que hace el programa con las cuatro primeras instrucciones es coger la dirección de la zona de la descripción de la cadena; si no tienes muy claro cómo se pasan parámetros a los programas, repasa en el capítulo 3 para aclarar tus dudas.

Continuando con el programa, en el par de registro BC está ahora la dirección del primer byte de la zona de la descripción, que es la longitud de la cadena. Después aumentamos el par BC para que señale al primer byte de la dirección real de la cadena. Las siguientes instrucciones hasta LD A,(HL) asignan dicha dirección en el par de registros HL. La instrucción LD se usa para conseguir el código del

primer carácter de la cadena. Después se modifica y se ejecuta un RETURN para volver al BASIC con la cadena ya modificada.

XOR

Con éste ya habremos visto todos los operadores lógicos. A pesar de su nombre (XOR), te aseguro que no es ningún personaje de una novela de... ¡ciencia-ficción!, sino que es la abreviatura de los términos ingleses: *eXclusive OR*.

Su tabla de verdad es la siguiente:

A	*B*	*A XOR B*
0	0	0
0	1	1
1	0	1
1	1	0

Como ves, la función XOR vale 1 cuando los dos bits sobre los que actúa son diferentes. Los modos de direccionamiento son los mismos que para la instrucción AND. La función XOR se utiliza, por poner un ejemplo, para asignar un 0 en el registro A; por ejemplo, la instrucción

```
XOR A
```

asigna 0 al registro A y solamente necesita 1 byte para codificarlo. De cualquier manera, la forma más común de cargar A con ceros es con la instrucción

```
LD A,0
```

que usa 2 bytes.

Efectos en los *flags*

Como puedes ver en la tabla al final de esta sección, este tipo de operaciones actúan, de un modo u otro, sobre casi todos los *flags*, excepto en tres de ellos, en los que el estado depende de los resultados de dichas instrucciones, que son:

Z

Este indicador se activa si el resultado es 0.

S

Este se activa sólo si el bit número 7 del resultado es 1.

P/V

El *flag* de paridad se activa si la paridad es par y se desactiva si es impar.

Gracias a las instrucciones booleanas que usan el registro A en sus operaciones, podemos manejar el *flag* de acarreo, como puedes ver en la tabla siguiente:

XOR A — se pone a 0 C, asigna un 0 en A.
OR A — se pone a 0 C, deja A como estaba.

Mnemónico	*Bytes*	*Tiempo*	*Efectos en los* flags *C*	*Z*	*P/V*	*S*	*N*	*H*
AND registro	1	4	0	*	*	*	0	1
AND número	2	7	0	*	*	*	0	1
AND (HL)	1	7	0	*	*	*	0	1
AND (IX + d)	3	19	0	*	*	*	0	1
AND (IY + d)	3	19	0	*	*	*	0	1
OR registro	1	4	0	*	*	*	0	1
OR número	2	7	0	*	*	*	0	1
OR (HL)	1	7	0	*	*	*	0	1
OR (IX + d)	3	19	0	*	*	*	0	1
OR (IY + d)	3	19	0	*	*	*	0	1
XOR registro	1	4	0	*	*	*	0	1
XOR número	2	7	0	*	*	*	0	1
XOR (HL)	1	7	0	*	*	*	0	1
XOR (IX + d)	3	19	0	*	*	*	0	1
XOR (IY + d)	3	19	0	*	*	*	0	1

Notación para los flags:

* indica que una operación ha afectado ese *flag*.
0 indica que el *flag* está a 0.
1 indica que el *flag* está a 1.
— indica que no ha sido afectado ese *flag*.

Cómo manejar bits

Dentro del conjunto de instrucciones del Z-80 hay dos que nos permiten manipular el estado de los bits dentro de un byte. Estas instrucciones se llaman SET y RESET, y casi todo lo que hacen éstas también lo hacen los operadores lógicos que acabamos de ver.

La instrucción SET, por ejemplo, fija un bit dado a 1. Tiene los siguientes modos de direccionamiento:

SET n,r
SET n,(HL)
SET n,(IX + d)
SET n,(IY + d)

donde *n* es el bit a fijar y va desde 0 a 7. Por ejemplo, con las instrucciones

```
LD  A,0
SET 0,A
```

el resultado del registro A sería un 1 y NO variarían los *flags*.

Las dos instrucciones anteriores también sirven para modificar los bits de un registro sin alterar los restantes. La instrucción complementaria a SET es la RESET, que hace cero cualquier bit que deseemos y actúa con los mismos modos de direccionamiento que la instrucción SET.

Por ejemplo:

```
LD     A,255
RESET 0,A
```

hace que el registro A contenga el valor 254. Para saber el estado de un bit dentro de un byte usa la instrucción BIT, que tiene los mismos modos de direccionamiento de las órdenes SET y RESET y actúa sobre el *flag* 0. Ejemplo:

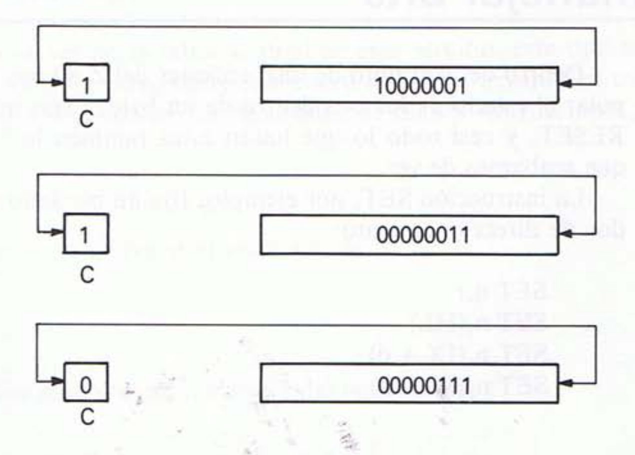
```
LD  IX,200
BIT 0,(IX + 0)
```

comprueba el bit 0 de la posición 200; si ese bit es 0, el *flag* 0 se activaría, pero si ese bit es 1 no se activará el *flag*.

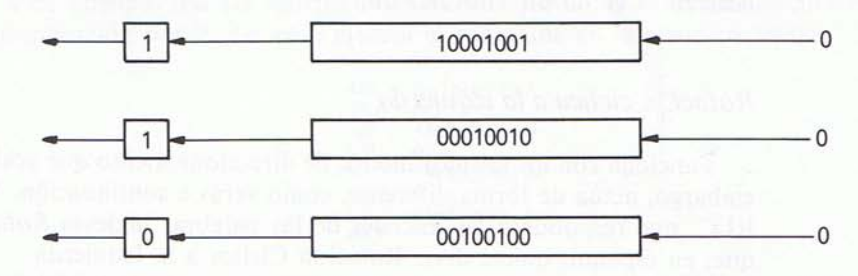
El último grupo de instrucciones que vamos a estudiar en este capítulo sobre instrucciones aritmético-lógicas con 8 bits es el de las instrucciones de ROTACION y DESPLAZAMIENTO.

Rotación y desplazamiento

Grosso modo, se puede decir que lo que hace una operación de rotación es llevar los bits en un sentido u otro dentro del mismo byte; de esta manera el bit primero terminaría el último, etc. Es mejor que te fijes en el dibujo para entender mejor esta operación cíclica:



Sin embargo, las operaciones de desplazamiento no son cíclicas:



Hay dos tipos de operaciones de rotación o de desplazamiento según el sentido en el que se muevan los bits: si los bits se mueven hacia la izquierda, se llama de rotación o de desplazamiento a la izquierda; si es hacia la derecha, se llama de rotación o de desplazamiento a la derecha.

Operaciones hacia la izquierda

Hay dos tipos diferentes de operaciones de rotación a la izquierda, y una sólo de desplazamiento a la izquierda. Veamos en qué consisten cada una.

Rotaciones a la izquierda

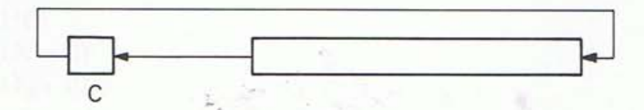
Esta operación actúa en los modos de direccionamiento siguientes:

RL r
RL (HL)
RL (IX + d)
RL (IY + d)

Un ejemplo práctico muy común puede ser

RL A

que es el acumulador de rotación a la izquierda. Lo que hace esta instrucción se puede representar con el gráfico siguiente:



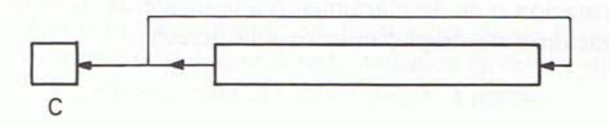
El valor del *flag* de acarreo se pone 0 y el bit número 7 se lleva al *flag;* así, el de acarreo es como un ''noveno'' bit.

Rotación cíclica a la izquierda

Funciona con los mismos modos de direccionamiento que acabamos de ver; sin embargo, actúa de forma diferente, como verás a continuación. Su mnemónico es RLC, que responde a las iniciales de las palabras inglesas *Rotate Left Circular,* que, en español, quiere decir Rotación Cíclica a la Izquierda.

Por ejemplo, instrucciones muy comunes son: RLC A y RLC (HL).

El diagrama de esta instrucción es como sigue:



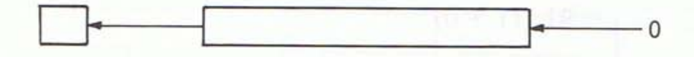
En este caso el valor del *flag* de acarreo no se mueve al bit 0 del byte.

Aritmética de desplazamiento a la izquierda

También trabaja con los mismos modos de direccionamiento que vimos anteriormente. Su mnemónico es

SLA s

donde *s* es cualquiera de los modos de direccionamiento. Su diagrama es como sigue:



Como ves, esta instrucción es como si fuera una multiplicación por 2; pero si el valor del acumulador es mayor que 127, después de ejecutar la orden SLA A, el resultado en el registro A será erróneo.

Un ejemplo práctico de lo anterior es el ejemplo 12.

EJEMPLO 12

Este ejemplo usa las instrucciones SLA A y XOR A y lo que ejecuta es una multiplicación por 2. Teclea y ejecuta el programa en la posición 58000:

```
LD   A,(&HF7F8)
SLA  A
LD   (&HF7F8),A
LD   A,2
LD   (&HF663),A
XOR  A
LD   (&HF7F9),A
RET
```

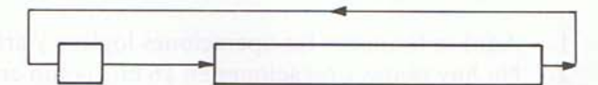
Ejecuta la rutina inicializándola con las instrucciones DEFUSR = 58000 y la instrucción PRINT USR(n). El resultado (N*2) se devolverá al BASIC.

Operaciones hacia la derecha

Las más simples son las de rotación hacia la derecha.

RR r
RR (HL)
RR (IX + d)
RR (IY + d)

El diagrama de esta instrucción sería:



Rotaciones cíclicas hacia la derecha

Usan idénticos modos de direccionamiento que las instrucciones RR, y su funcionamiento lo entenderás mejor con el siguiente dibujo:

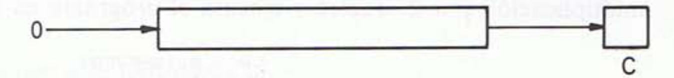


Desplazamiento lógico hacia la derecha

Los modos de direccionamiento que estas instrucciones utilizan son:

SRL r
SRL (HL)
SRL (IX + d)
SRL (IY + d)

y lo que en realidad ejecutan son divisiones por 2; el bit 7 se pone a 0 y el bit 0 va al acarreo:

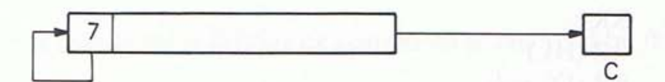


Un problema con el que se encuentran estas instrucciones, al ser como divisores por 2, es que necesitan saber dónde está el número en complemento a dos; en caso de que haya un número, es muy importante retener el bit de signo, es decir, el bit número 7.

La instrucción que se encarga de desplazar el bit de signo es la operación de desplazamiento aritmética.

Desplazamiento a la derecha aritmético

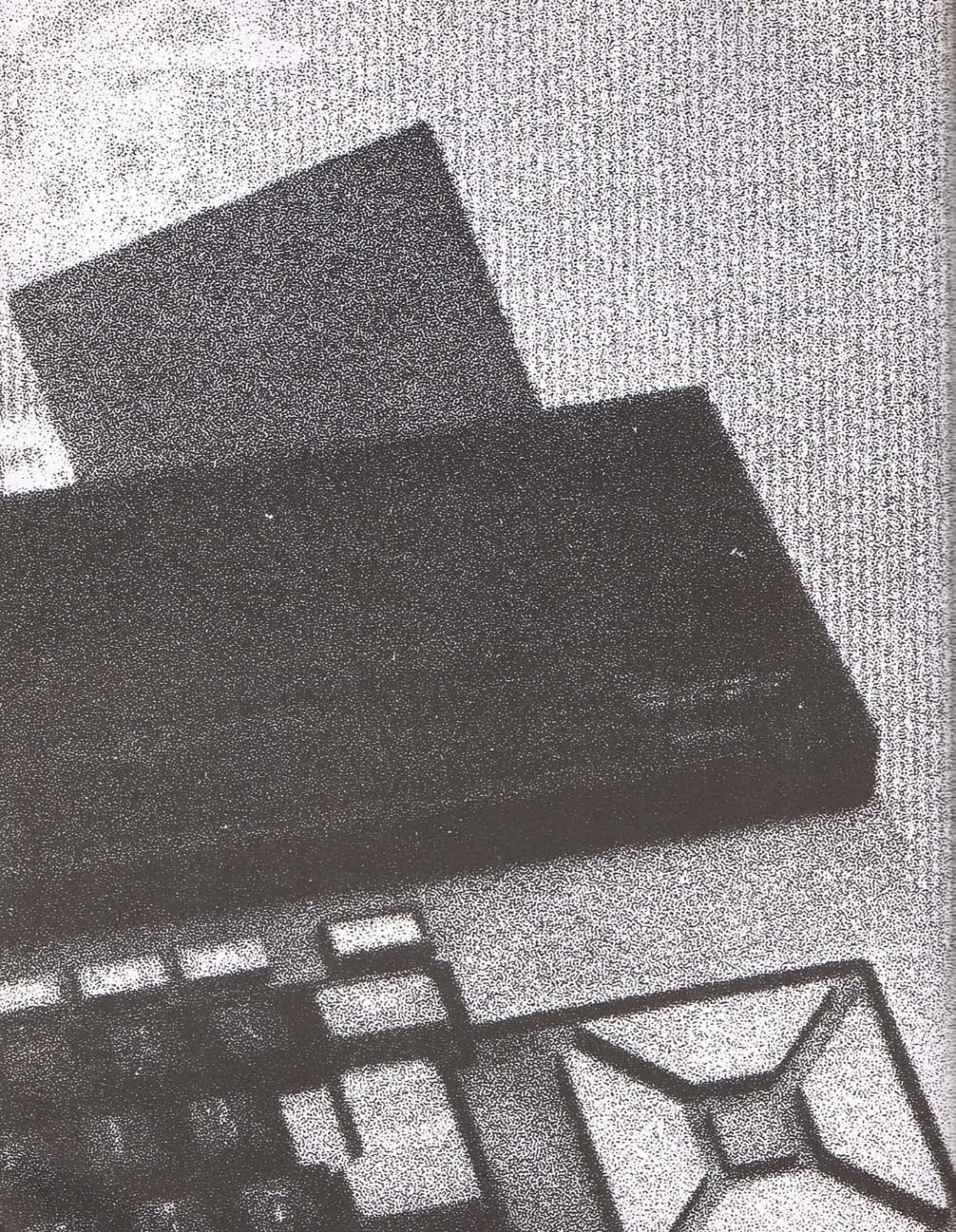
No desplaza el valor del bit 7 en ese momento; por todo lo demás, se comporta igual que la orden SRL. Su mnemónico es SRA s, y su diagrama, el siguiente:



Seguro que aquellos de vosotros que seguís intentando aprender conmigo este lenguaje máquina estáis de suerte, porque...

1. Aquí se terminan las operaciones lógicas y aritméticas con 8 bits.
2. No hay tantas operaciones en 16 bits como en 8 bits.

En los dos capítulos siguientes, veremos las instrucciones para transferencias de datos de 16 bits y las operaciones lógicas y aritméticas también con 16 bits.



6

Transferencia de datos de 16 bits

Ya has visto cómo unir dos registros de 8 bits para formar un PAR DE REGISTROS de 16 bits, con lo que ya puedes manejar números de hasta 16 bits. En este capítulo estudiaremos cómo pasar números de 16 bits entre la UCP y la memoria.

En la programación del lenguaje máquina Z-80 los números de 16 bits sirven para especificar la dirección de un número de 8 bits. En realidad ya hemos visto su uso en la práctica; las instrucciones tienen el siguiente formato:

LD A,(nn)

Pasemos ahora a estudiar estas instrucciones en profundidad. Primero vamos a ver aquellas instrucciones que se encargan de cargar o asignar un número de 16 bits a un registro de 16 bits. El mnemónico de esta instrucción es:

LD rr,nn

donde *rr* es un par de registros de 16 bits y *nn* un número de 16 bits. Ten en cuenta que, si especificas un número de 16 bits en un programa de código máquina, primero debes dar el byte bajo; por ejemplo, la instrucción

LD HL,&HF7F8 se codifica 21 F8 F7

De forma parecida se carga también información directamente en los registros IX e IY.

Transferencias entre pares de registros y la memoria

Las instrucciones para esta función son:

```
LD rr,(nn)
LD IX,(nn)
LD IY,(nn)
```

donde *nn* es una dirección de 16 bits. Lo que estamos haciendo es cargar el contenido de la dirección en el par de registros de 16 bits. Debido a que estamos trabajando con números de 16 bits, lo que en realidad estamos haciendo es cargar el par de registros, los contenidos de la dirección *nn* y la dirección siguiente. En el ejemplo siguiente:

```
LD HL, (1000)
```

el registro L es en este caso el byte bajo del registro y tendría el contenido de la dirección 1000, y el registro H, el contenido de la posición 1001. Por tanto, no tienes que especificar ambas direcciones cuando cargues números de 16 bits.

Para hacer la operación contraria, es decir, cargar posiciones de memoria de registros de 16 bits, utilizaremos las instrucciones siguientes:

```
LD (nn),rr
LD (nn),IX
LD (nn),IY
```

Ya hemos visto ejemplos que usan estas instrucciones anteriormente; fíjate en el siguiente:

```
LD HL,&HF7F8
LD A,(HL)
```

Lo que hacen estas instrucciones es cargar el registro A de la posición dada por el par de registros HL. De la misma manera,

```
LD BC,(&HF7F8)
```

cargaría el contenido de la posición &HF7F8 en el registro C y el contenido de la posición &HF7F9 en el registro B. Esta instrucción es muy útil, ya que consigue recoger fácilmente un entero que se haya pasado a una rutina en código máquina, como parámetro de la función USR a un par de registros. Las instrucciones *LoaD* (para cargar) tienen códigos *op* de 4 bytes: dos bytes representan la dirección a la que se va a acceder y los otros dos bytes representan la instrucción.

Mnemónico	Bytes	Tiempo	Efectos en los flags C	Z	P/V	S	N	H
LD Par de Registros, Número	3/4	10	—	—	—	—	—	—
LD IX, Número	4	14	—	—	—	—	—	—
LD IY, Número	4	14	—	—	—	—	—	—
LD (Dirección), BC o DE	4	20	—	—	—	—	—	—
LD (Dirección), HL	3	16	—	—	—	—	—	—
LD (Dirección), IX	4	20	—	—	—	—	—	—
LD (Dirección), IY	4	20	—	—	—	—	—	—
LD BC o DE, (Dirección)	4	20	—	—	—	—	—	—
LD HL, (Dirección)	3	16	—	—	—	—	—	—
LD (IX, (Dirección)	4	20	—	—	—	—	—	—
LD IY, (Dirección)	4	20	—	—	—	—	—	—

Notación para los flags:

- * indica que una operación ha afectado ese *flag*.
- 0 indica que el *flag* está a 0.
- 1 indica que el *flag* está a 1.
- — indica que no ha sido afectado ese *flag*.

Cómo usar la pila

Como recordarás, ya te expliqué lo que era una pila y también te dije que una pila es donde la UCP almacena información, y que además no necesitas recordar dónde la dejaste. Sin embargo, sólo podemos almacenar números de 16 bits en la pila. Las instrucciones que almacenan registros en la pila son las siguientes:

```
PUSH AF
PUSH rr
PUSH IX
PUSH IY
```

donde *rr* es cualquier par de registros; date cuenta de que estamos usando el par de registros AF como uno de 16 bits.

Ya dije que todas estas instrucciones PUSH servían para "empujar" o añadir información dentro de la pila; tienen un código de operación, o código *op,* de un solo byte. Las instrucciones PUSH copian el contenido del par de registros de 16 bits que queramos en la pila. Debido a que éstas son operaciones que copian un par de registros, después de copiarse, éste seguirá teniendo el número almacenado en la pila. Para sacar un número de una pila se usan las instrucciones POP, que sacan de la pila el último número que se añadió en ella con una instrucción PUSH.

```
POP AF
POP rr
POP IX
POP IY
```

También podemos usar las pilas para hacer transferencias entre registros de 16 bits, como, por ejemplo:

```
LD BC,DE
```

Esta función no está implementada dentro del conjunto de instrucciones del Z-80 y normalmente se tienen que usar las dos instrucciones equivalentes a ésta, que son:

```
LD B,D
LD C,E
```

o, más elegantemente, las instrucciones

```
PUSH DE
POP  BC
```

Para saber directamente el contenido del registro F, usamos también dos instrucciones: una PUSH y otra POP.

```
PUSH AF
POP  BC
```

El registro BC tiene ahora el contenido que tenía el registro AF, por tanto, el registro C será el que tiene ahora el contenido del F.

Otro uso alternativo del registro F sería para almacenar los valores del *flag* de registro en ese momento mientras se terminan otras operaciones.

```
PUSH AF
```

La otra instrucción sería:

```
POP  AF
```

Nota: El sistema operativo del ordenador usa la pila para lo mismo que nosotros la utilizamos. Cuando ejecutamos una instrucción USR, la dirección desde la cual se hizo la llamada, y a la que la UCP volverá después de ejecutar nuestro código máquina, se añade en la pila. Es muy importante que esa dirección sea la primera a la que se acceda de la pila cuando el programa en código máquina del usuario ejecute la instrucción RET.

Esta instrucción hace que la UCP vuelva a la primera dirección de la pila; como ves, todas las instrucciones PUSH deben tener sus correspondientes POP. Por ejemplo:

```
POP  HL
RET
```

causaría problemas a la hora de ejecutarlo, pero, sin embargo,

```
PUSH HL
POP  BC
RET
```

sería correcto, ya que la instrucción PUSH tiene su correspondiente POP.

Cómo mover la pila

Aunque normalmente no necesitemos saber el paradero de la pila en la memoria, lo primero que la UCP tiene que hacer, mientras esté bajo el control del sistema operativo MSX, es inicializar la pila. La UCP carga la dirección en la que quiere poner la pila en un registro de 16 bits llamado PUNTERO de la pila o SP (*Stack Pointer,* que quiere decir puntero de la pila). Las instrucciones para manejar la posición de la pila dentro de la RAM son:

```
LD SP,nn
LD SP,(nn)
LD SP,IX
LD SP,IY
```

Una advertencia antes de continuar: como es lógico, mover la pila por la memoria es no sólo una pérdida de tiempo, sino que también podría causar que la UCP perdiese la dirección a la que tiene que volver cuando se haya ejecutado la rutina del usuario en código máquina.

El último grupo de instrucciones sobre transferencias de 16 bits que vamos a ver es el de las que lo hacen entre el conjunto de registros principales, es decir, los registros que hemos usado hasta ahora, y el GRUPO DE REGISTROS ALTERNATIVOS o CONJUNTO DE REGISTROS PRIMARIOS, que son:

AF', BC', DE', HL'.

No hay registros alternativos para IX, IY y SP. Estos registros, llamémoslos alternativos, se usan normalmente como almacenes temporales del contenido de los registros principales. Las instrucciones LD no afectan a estos registros y no se pueden realizar operaciones aritméticas con este grupo de registros. Para lo único que sirven es para intercambiar datos entre éstos y los anteriores, usando las instrucciones de INTERCAMBIO DE REGISTROS.

La instrucción

```
EX AF,AF'
```

intercambia el contenido del registro AF con el de su alternativo.

La instrucción

EXX

hace lo mismo simultáneamente con los pares de registros BC, DE y HL. La última instrucción de este tipo intercambia dos pares de registros del grupo principal de registros; por ejemplo, la instrucción

EX DE,HL

intercambia el contenido del par de registros DE con el HL.

Con esta última explicación quedan completas las instrucciones que transfieren datos de 16 bits.

Mnemónico	*Bytes*	*Tiempo*	*Efectos en los* flags					
			C	*Z*	*P/V*	*S*	*N*	*H*
PUSH Par de Registros	1	11	—	—	—	—	—	—
PUSH IX o IY	2	15	—	—	—	—	—	—
POP Par de Registros	1	10	—	—	—	—	—	—
POP IX o IY	2	14	—	—	—	—	—	—
LD SP, Dirección	3	10	—	—	—	—	—	—
LD SP,(Dirección)	3	20	—	—	—	—	—	—
LD SP, HL	1	6	—	—	—	—	—	—
LD SP, IX o IY	2	10	—	—	—	—	—	—

Notación para los flags:

* indica que una operación ha afectado ese *flag*.
0 indica que el *flag* está a 0.
1 indica que el *flag* está a 1.
— indica que no ha sido afectado ese *flag*.

EJEMPLO 13

Las posiciones 64670 y 64671 tienen el valor de la función TIME del BASIC. Este ejemplo que ahora vamos a ver usa las instrucciones que vimos antes para asignar las posiciones anteriores a 0. Si ejecutas PRINT USR (0) o L = USR (0), hacen lo mismo que TIME = 0.

Introduce el programa siguiente a partir de la dirección 58000; y usa la instrucción DEFUSR = 58000 para definir la rutina.

```
LD    BC,00                  01   00   00
LD    (64670),BC             ED   43   9E   FC
RET                          C9
```

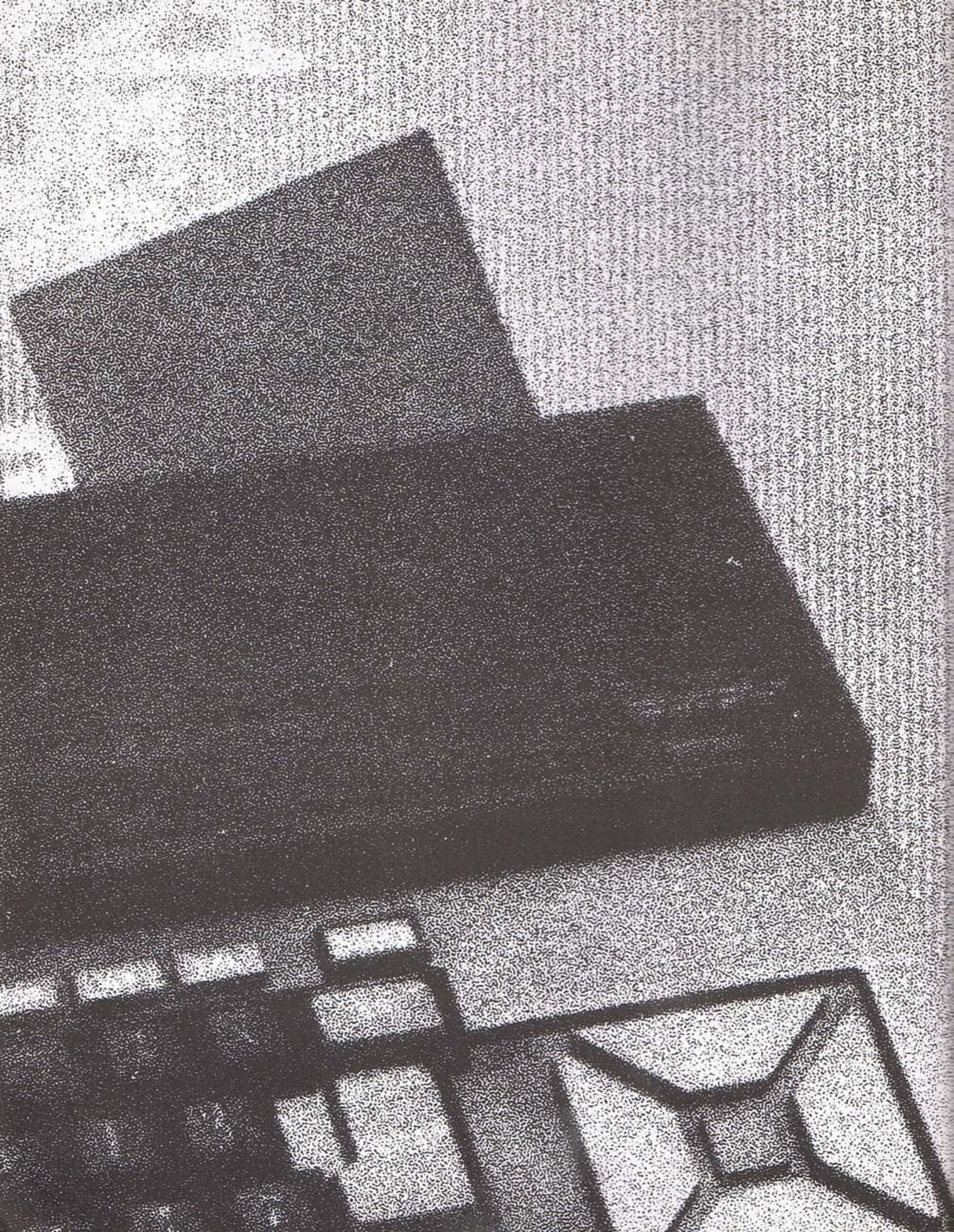
Después ejecuta la siguiente instrucción en modo directo:

TIME = 1200 : 1 = USR(0) : PRINT TIME

EJEMPLO 14

La rutina siguiente coge un valor entero que se le ha pasado como parámetro; después intercambia el bit alto y el bajo y luego devuelve este entero una vez cambiado. En este ejemplo también utilizo pares de registros de 16 bits.

```
LD    BC,(&HF7F8)
LD    D,C
LD    E,B
LD    (&HF7F8),DE
LD    A,2
LD    (&HF663),A
RET
```



7

Cálculos y aritmética con 16 bits

Las instrucciones aritméticas de 16 bits sirven para hacer más fáciles las operaciones de suma y resta en 16 bits; ya hemos visto cómo se usan las instrucciones ADC con registros de 8 bits para realizar cálculos aritméticos con varios bytes. Pero aunque las instrucciones de 16 bits para estos cálculos sean más fáciles de usar, no son tan versátiles como las de 8 bits.

Como primer ejemplo sencillo en 16 bits, vamos a ver cómo se resta o suma un 1 a un número de 16 bits. Dicho cálculo se hace, como recordarás, con las instrucciones INC y DEC.

Cómo incrementar y decrementar

La instrucción más sencilla para aumentar un número es:

```
INC   rr
```

que aumentará el contenido de un par de registros. En las instrucciones siguientes:

```
LD    HL,0000
INC   HL
```

el resultado será un 1 en el par de registros HL. También podemos modificar los registros IX e IY.

```
INC  IX
INC  IY
```

Las instrucciones DEC son análogas a las anteriores. Es decir:

```
DEC  BC
DEC  DE
DEC  HL
DEC  IX
DEC  IY
```

La diferencia más importante entre las instrucciones INC y DEC en 8 bits o en 16 es que los cálculos con las instrucciones de 16 bits NO afectan para nada a los *flags;* lo cual es un gran inconveniente, ya que necesitaremos otras instrucciones, al margen de las anteriores, para saber el valor de cualquier registro de 16 bits. Por ejemplo: para saber si un par de registros es igual a 0, puedes usar el siguiente programa:

```
DEC  HL
LD   A,L
OR   H
JP   Z,59000
```

En este pequeño programa he usado la instrucción JP Z para controlar que el programa vaya a la dirección 59000 si el resultado de la comparación es 0. Si tanto L como H están a 0, entonces la instrucción OR activará el *flag* 0.

Operaciones de adición y sustracción

Al igual que en 8 bits el registro A es un tanto especial, en los cálculos aritméticos en 16 bits el par de registros "especial" es el HL. Las instrucciones ADD *(ADDition)* para pares de registros de 16 bits son las siguientes:

```
ADD  HL,rr
ADD  IX,BC
ADD  IX,DE
ADD  IX,SP
ADD  IX,IX
ADD  HL,SP
ADD  IY,BC
ADD  IY,DE
ADD  IY,SP
```

Si te fijas un poco en la lista de instrucciones anterior, te darás cuenta de dos cosas: la primera es que no existe ninguna instrucción que sume el contenido del par de registros HL a ninguno de los registros índices, y la segunda es que no hay ninguna instrucción del tipo

ADD IX,IY o ADD IY,IX

Finalmente, date cuenta de que tampoco existe la instrucción

ADD HL,nn

Para poder realizar la operación anterior, se hace lo siguiente:

```
LD   DE,nn
ADD  HL,DE
```

que presenta la desventaja de tener que utilizar dos pares de registros.

En todos estos cálculos, el resultado se deja en el par de registros determinado en la instrucción correspondiente; por ejemplo, en la instrucción

ADD HL,BC

el resultado se deja en el par de registros HL.

Efectos en los *flags*

No hay casi ningún *flag* que sea afectado por los cálculos aritméticos de 16 bits. De los *flags,* el C se activará si hay acarreo del MSB (bit más significativo) del registro H, es decir, del bit ''decimoséptimo'' del registro HL. Cualquier acarreo del MSB del registro L será llevado por esa misma instrucción al registro H. El otro *flag* afectado es el N, que se fijará a 0.

Sumas y restas con acarreo

Las instrucciones ADC, igual que en 8 bits, son capaces de ejecutar cálculos aritméticos con varios bytes. Las instrucciones ADC son:

```
ADC  HL,BC
ADC  HL,DE
ADC  HL,HL
ADC  HL,SP
```

Como verás, no hay ninguna instrucción que trabajen con los registros INDICES. No hay tampoco instrucciones SUB, es decir, restas sin acarreo, por lo que tendrás que usar las instrucciones SBC para realizar dichos cálculos; las instrucciones SBC son las siguientes:

```
SBC  HL,BC
SBC  HL,DE
SBC  HL,HL
SBC  HL,SD
```

Estas instrucciones toman en cuenta el *flag* de acarreo en sus cálculos. Recuerda que debes inicializar dicho *flag* cuando uses esas instrucciones para restas sencillas en 16 bits; la forma más fácil de hacerlo es usando algunas de las funciones booleanas de 8 bits:

```
LD   HL,(58000)
LD   DE,(58002)
AND  A
SBC  HL,DE
LD   (58000),DE
RET
```

La rutina anterior resta el contenido de la posición 58002 al de la posición 58000. La instrucción AND A inicializa el *flag* C. Observa en la tabla siguiente que las instrucciones ADC y SBC también afectan a los *flags* Z, P/V y S, de acuerdo con el resultado de una operación.

Mnemónico	*Bytes*	*Tiempo*	*Efectos en los* flags C	Z	P/V	S	N	H
ADD HL, Par de Registros	1	11	*	—	—	—	0	?
ADD HL, SP	2	11	*	—	—	—	0	?
ADC HL, Par de Registros	2	15	*	*	*	*	0	?
ADC IX, SP	2	15	*	*	*	*	0	?
ADD IX, BC o DE	2	15	*	—	—	—	0	?
ADD IX, IX	2	15	*	—	—	—	0	?
ADD IX, SP	2	15	*	—	—	—	0	?
ADD IY, BC o DE	2	15	*	—	—	—	0	?
ADD IY, IY	2	15	*	—	—	—	0	?
ADD IY, SP	2	15	*	—	—	—	0	?
SBC HL, Par de Registros	2	15	*	*	*	*	1	?
SBC HL, SP	2	15	*	*	*	*	1	?

Notación para los flags:

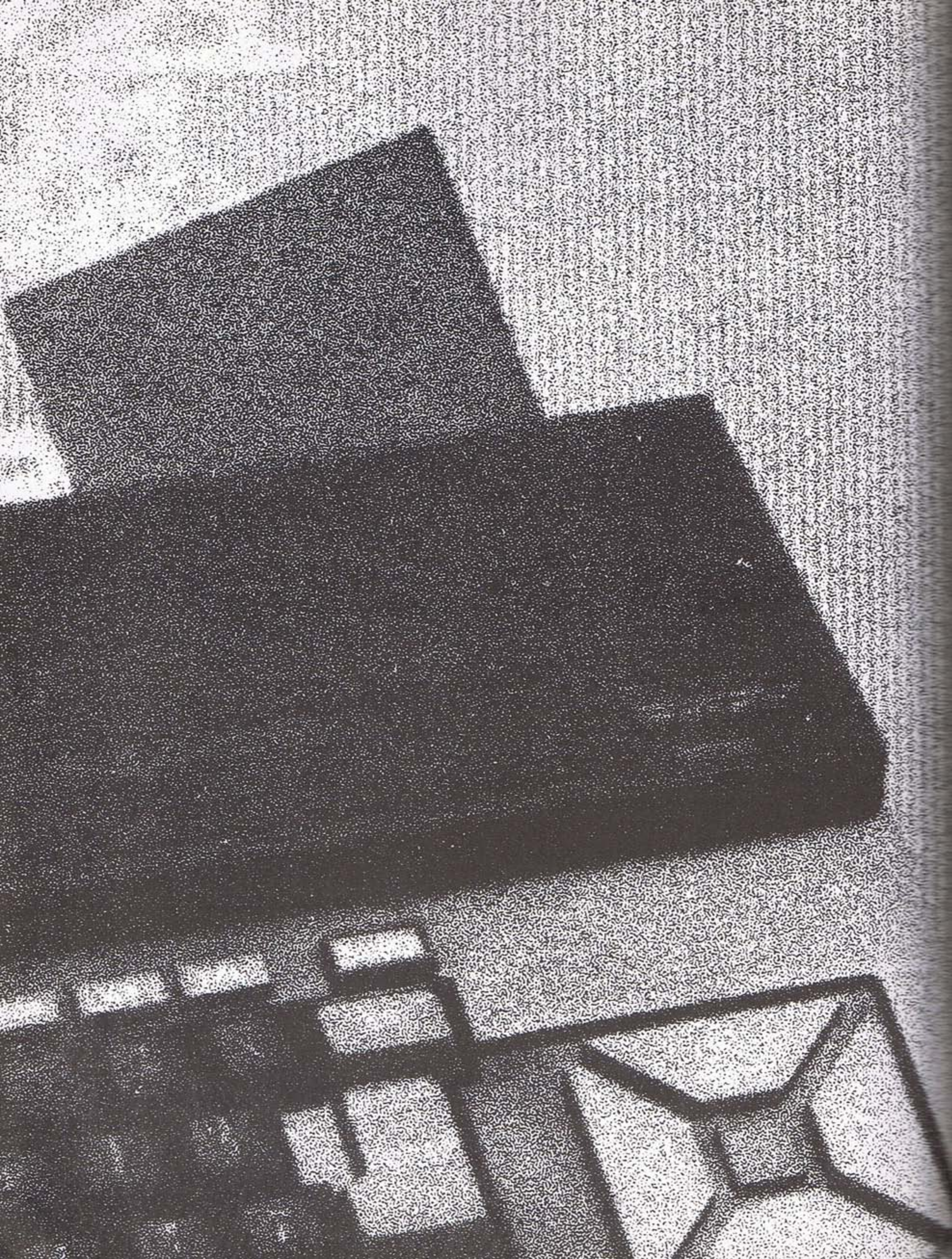
- * indica que una operación ha afectado ese *flag*.
- 0 indica que el *flag* está a 0.
- 1 indica que el *flag* está a 1.
- — indica que no ha sido afectado ese *flag*.
- ? indica que no se conoce el efecto.

EJEMPLO 15

Este ejemplo es una suma de 16 bits muy sencilla. Almacena este programa a partir de la dirección 58000; el programa sumará 500 a cualquier valor que se pase a la rutina como parámetro entero.

Fíjate que, si el número es mayor que 32767, esta operación devolverá al BASIC un número negativo, es decir, un número en complemento a dos.

```
LD   BC,500
LD   HL,(&HF7F8)
ADD  HL,BC
LD   (&HF7F8),HL
LD   A,2
LD   (&HF663),A
RET
```



serían cada vez menos eficientes. Si usas las instrucciones en código máquina equivalentes a la "GOTO", tus programas serán mucho más potentes, pero ten en cuenta que debes usar estas instrucciones con precaución.

Empecemos con las instrucciones JP, abreviatura de la palabra inglesa *JumP*, que, en español, quiere decir "salto", y que son el equivalente más directo del GOTO del BASIC.

La instrucción JP tiene dos modos de direccionamiento: inmediato y registro indirecto. En el modo inmediato, la dirección a la que se va a pasar el control del programa viene especificada implícitamente en la instrucción; como, por ejemplo:

```
JP 58000
```

En el modo de registro indirecto, la dirección a la que se va a pasar el control se almacena en cualquiera de los siguientes pares de registros: en el IX o en el IY o bien en el HL.

Más adelante y en este mismo capítulo te explicaré esta instrucción con más detenimiento.

Un salto puede ser INCONDICIONAL, como el de antes, en el que el control se pasa inmediatamente a la dirección especificada; o también puede ser CONDICIONAL, que son aquellos saltos que sólo se realizan si se cumplen determinadas condiciones. Como ves, ésta sí es la instrucción en código máquina equivalente a la del BASIC:

```
IF ... THEN ... GOTO
```

Por ejemplo, la instrucción

```
JP Z,59000
```

saltará a la posición 59000 si el *flag* Z está a 1. También se pueden usar otros *flags* para decidir si se tiene que realizar un salto o no; por ejemplo:

```
JP C,59000
```

ejecutará un salto a la dirección 59000 cuando se active el *flag* C. Otras instrucciones de este mismo tipo son:

```
JP NZ  salta si el resultado no ha sido 0
JP NC  salta si no se ha producido acarreo
JP P   salta si el resultado es positivo
JP M   salta si el resultado es negativo
JP PE  salta si la paridad es par
JP PO  salta si la paridad es impar
```

Todas las instrucciones anteriores ocupan tres bytes: un byte para el código de operación y los otros dos para la dirección. Como ya bien imaginas, primero se almacena el byte bajo de la dirección; ejemplo:

JP Z, E290 se codifica como CA 90 E2

Veamos ahora un programa en código máquina sencillo que use las instrucciones JP. Pero, antes de empezar, lee atentamente estas dos advertencias:

a) Cuando le ordenes a la UCP que salte, ella saltará aunque te hayas confundido de dirección, y, como es lógico, el salto resultante puede ser a una dirección que contenga un dato o que sea el segundo byte de una instrucción de varios bytes. Como ves, la mayoría de las veces que te equivoques el programa no funcionará.

b) Si el salto hace que una instrucción se ejecute siempre, como, por ejemplo, la de pasarse el control a sí misma, no hay forma de recuperar el control, excepto pulsando la tecla... ¡RESET!

Siguiendo con el ejemplo que te dije antes, a continuación te doy la lista de los bytes para que el salto sea correcto.

EJEMPLO 16

Este programa borra la pantalla en modo 0:

La instrucción

1 = USR(0)

llena la pantalla con caracteres CHR$(0).

La instrucción

1 = USR(65)

llena la pantalla con caracteres A. El parámetro que se pasa siempre es un entero. Como ya es habitual, escribe el programa en la posición 58000 y no olvides usar la instrucción DEFUSR para determinar la dirección de la instrucción USR.

Debes teclear la dirección del salto con el byte bajo primero.

```
        LD   HL,960          21  C0  03
        IN   A,(&H99)        DB  99
        LD   A,0             3E  00
        OUT  (&H99),A        D3  99
        LD   A,64            3E  40
        OUT  (&H99),A        D3  99
BUCLE   LD   A,(&HF7F8)      3A  F8  F7
        OUT  (&H98),A        D3  98
        DEC  HL              2B
        LD   A,L             7D
        OR   H               B4
        JP   NZ,BUCLE        C2  9D  E2
        RET                  C9
```

Si te fijas, he usado una etiqueta *(label)* para decirle a la instrucción JP dónde tiene que pasar el control del programa: esto lo hago simplemente para mayor comodidad y, así, cuando ensamblo el programa pongo la dirección correcta. En este caso particular, la instrucción JP pasa el control a la dirección &HE290.

El código debes ponerlo en la dirección 58000; de otro modo, la dirección de la instrucción JP sería diferente a la que tú quieres. Pasemos ahora a ver esta parte del programa:

La primera instrucción cargará el número de bytes de la pantalla cuando está en modo 0 en el par de registros HL; este registro lo usaremos como contador del programa. Las siguientes cinco instrucciones asignan la dirección del primer byte de la VRAM donde quieres escribir; esto se consigue escribiendo la dirección codificada en los registros VDP.

La dirección anterior es la posición 0 de la VRAM. En otro capítulo te explicaré con más detalle cómo las direcciones de la VRAM se escriben en el VDP. Vayamos ahora a estudiar la etiqueta BUCLE; la instrucción LD A coge de la zona de parámetro el carácter en código ASCII con el que vamos a llenar la pantalla; después se escribe este carácter en el VDP que se encarga de poner el carácter ASCII en el primer byte de la VRAM, con lo que también se consigue de manera muy inteligente que el VDP señale el segundo byte de la VRAM; este proceso se repite 960 veces, con lo que la pantalla quedará rellena de ese carácter.

Cuando ejecutes este programa, te darás cuenta de la asombrosa velocidad de los programas en código máquina.

EJEMPLO 17

A veces la velocidad de ejecución de un lenguaje en código máquina es demasiado rápida para algunas aplicaciones, por lo que quizá tengas que crearte un bucle de retardo para disminuir la velocidad de ejecución de los programas; este ejemplo te enseña cómo usar esos bucles para disminuir la velocidad de tus rutinas.

```
       LD   HL,65535      21  FF  FF
BUCLE  LD   DE,100        11  64  00
BUCLE2 DEC  DE            1B
       LD   A,E           7B
       OR   D             B2
       JP   NZ,BUCLE2     C2  96  E2
       DEC  HL            2B
       LD   A,H           7C
       OR   L             B5
       JP   NZ,BUCLE      C2  93  E2
       RET                C9
```

Debes introducirlo en la posición acostumbrada: la 58000. En este programa aparece de nuevo la instrucción OR, que comprueba si está a 0 el contador; en este caso, sólo el registro L.

Como ves, con estas instrucciones JP podemos hacer programas cuyo funcionamiento dependa del estado de los *flags*. Las instrucciones JP que hemos visto

hasta ahora requieren que el programador especifique la dirección con 2 bytes, incluso si el salto de destino sólo estuviera a unas pocas instrucciones del de origen.

También existen unos saltos relativos que sólo necesitan un byte para especificar la dirección de destino; dichos saltos se llaman SALTOS RELATIVOS.

Saltos relativos

En los ejemplos que hemos visto hasta aquí, aunque la dirección de destino de los saltos no estaba muy lejos de la instrucción JP, siempre teníamos que usar 2 bytes para indicar la dirección de destino. Además, usar una dirección concreta en las instrucciones JP significaba que nuestros programas siempre tenían que ser ejecutados en la misma parte de la memoria, es decir, donde el programa originariamente se escribió, debido a que especificaste la dirección a la que iba a saltar la instrucción JP.

Sin embargo, con la instrucción de salto relativo se evitan dichos problemas. Esas instrucciones sólo ocupan 2 bytes: 1 byte en código *op* y el otro para la dirección a la cual se pasará el control una vez ejecutado el salto. Con este último byte, es decir, con el de desplazamiento, puede representarse un número entre —128 y + 127; dicho número es la distancia a la que va a saltar la instrucción. Como verás, podemos pasar el control a cualquier byte, hasta 127 bytes después del salto relativo y hasta 128 bytes antes del salto relativo.

El mnemónico para esta instrucción es:

JR cc,d

donde *cc* es una de las condiciones aplicables a los saltos relativos y *d* es el byte de desplazamiento. También tenemos saltos relativos incondicionales, del tipo

JR d

El valor del byte de desplazamiento provoca un salto de la siguiente manera: cuando la UCP se encuentra una instrucción JR d o una JR cc, lo que ocurre es que se añade un 2 al valor del contador del programa. La dirección del salto se calcula sumando la dirección del byte que está después de la instrucción JR y del byte de desplazamiento:

```
INC  A      —3  INC  A
JR   Z,02   —2  JR   Z,
LD   A,0     0  LD   A,
LD   A,2    +1       00
            +2  LD   A,
            +3       02
```

El byte que está después del de desplazamiento es el 0, el siguiente es el + 1, etcétera. De igual modo, el byte que está antes del byte de desplazamiento es el — 1, la instrucción JR será el — 2, etc. Así, la instrucción

JR — 02

no es una buena idea, ya que hace que la UCP ejecute la instrucción JR en un bucle infinito.

Como es lógico, los números negativos se almacenan en complemento a dos; así, por ejemplo, el número — 02 sería el &HFE. El siguiente ejemplo es algo más complicado e incluye una etiqueta:

```
         LD    A,00
BUCLE    INC   A
         JR    Z,BUCLE
```

El byte de desplazamiento que he puesto después de la instrucción JR en este ejemplo es un — 3, que en el programa se escribe como &HFD.

No todas las condiciones están implementadas en las instrucciones JR; las que están a tu disposición son:

JR C,d
JR NC,d
JR Z,d
JR NZ,d

Por lo que, si quieres, por ejemplo, hacer un salto condicionado por la paridad y el signo del resultado de una operación, tendrás que usar las instrucciones JP. La mayor ventaja que ofrecen las instrucciones de salto relativo es que no hacen referencia a ninguna posición determinada de la memoria; todas las posiciones de memoria se toman con respecto a la posición de la instrucción JR en ese momento. Esto quiere decir que un programa que haya sido escrito con saltos relativos puede ejecutarse en cualquier posición de memoria; fíjate en el ejemplo 18 para ver esto en la práctica.

EJEMPLO 18

Este ejemplo es una repetición del 16, pero con saltos relativos, en vez de los absolutos:

```
        LD    HL,960          21  C0  03
        IN    A,(&H99)        DB  99
        LD    A,0             3E  00
        OUT   (&H99),A        D3  99
        LD    A,64            3E  40
        OUT   (&H99),A        D3  99
BUCLE   LD    A,(&HF7F8)      3A  F8  F7
        OUT   (&H98),A        D3  98
        DEC   HL              2B
        LD    A,L             7D
        OR    H               B4
        JR    NZ,BUCLE        20  F6
        RET                   C9
```

Teclea el código, como antes, en la posición 58000 y después ejecútalo. Luego usa la opción MOVE para llevar el código a la dirección 59000. Cambia la instrucción DEFUSR para que señale esta nueva dirección y seguidamente ejecuta el código en la misma posición.

El caso 18 es un ejemplo de código reubicable, es decir, puedes ejecutarlo en una dirección diferente a la que fue escrito.

Saltos indirectos a través de registros

Antes ya he mencionado este tipo de saltos; lo que tienes que hacer es asignar la dirección a la que quieres saltar en cualquiera de los registros siguientes: HL, IX e IY, y ejecutar después la instrucción correspondiente:

```
JP (HL)
JP (IX)
JP (IY)
```

Por ejemplo, las instrucciones siguientes pasan el control a la posición 0 del ordenador, que en los ordenadores MSX es la rutina de RESET.

```
LD  HL,0000
JP  (HL)
```

Bucles FOR ... NEXT en código máquina

Hasta aquí sólo hemos visto las instrucciones en código máquina equivalentes a las IF ... THEN del BASIC; ahora vamos a ver las equivalentes al bucle FOR ... NEXT, también del BASIC. Normalmente este bucle se usa cuando se puede especificar las veces que se quiere ejecutar determinada secuencia del programa. Veamos ahora un simulacro en código máquina del programa en BASIC siguiente:



```
10 C=0
20 FOR I=1 TO 6
30 LET C=C+1
40 NEXT I
```

En lenguaje máquina vas a usar un registro en vez de la variable I. La manera más fácil de hacer el programa anterior con un sólo registro es haciendo una cuenta atrás para comprobar si el contenido del registro es 0:

```
        LD     C,0       inicializa C
        LD     B,6       inicializa "I"
BUCLE   INC    C         c = c + 1
        DEC    B         esta instrucción y la siguiente simulan la
                         instrucción NEXT del BASIC
        JR     NZ,BUCLE
```

Las últimas dos instrucciones del programa aparecen con frecuencia juntas en los programas en código máquina Z-80, por lo que han sido unificadas en una sola instrucción: la DJNZ. La sintaxis completa de esta instrucción es la siguiente:

```
DJNZ desplazamiento
```

donde "desplazamiento" es el byte de desplazamiento igual al que vimos en las instrucciones de saltos relativos. Dicha instrucción es la sigla correspondiente a las siguientes palabras inglesas: *Decrement B register and Jump if Not Zero;* en español significa: disminuye el registro B y ejecuta el salto si no es cero dicho registro.

Esta instrucción necesita dos bytes, por lo que se ahorra el byte que tendrías que utilizar con la misma instrucción pero separada, es decir, con las instrucciones DEC y JR NZ.

El programa anterior se puede escribir como sigue:

```
        LD     C,0
        LD     B,6
BUCLE   INC    C
        DJNZ   BUCLE
```

Hay un solo problema con esta instrucción DJNZ, y es que sólo se puede usar para ejecutar bucles hasta 256 veces; no hay instrucciones DJNZ de 16 bits. Como he dicho, sólo podemos pasar por ese bucle 256 veces; pero ¿cómo es esto posible si el mayor número que puede contar un registro de 8 bits es el 255? La respuesta es que, si asignas un 0 al registro B y después ejecutas la instrucción DJNZ, el registro B disminuye y entonces queda 255.

Las instrucciones DJNZ se pueden anidar para que podamos ejecutar bucles de instrucciones más de 256 veces seguidas; fíjate en el ejemplo siguiente:

```
                  LD     B,16
BUCLE EXTERIOR    PUSH   BC
                  LD     B,256
BUCLE INTERIOR    ...
                  ...
                  DJNZ   BUCLE INTERIOR
                  POP    BC
                  DJNZ   BUCLE EXTERIOR
```

Otro programa parecido es el que usa registros de 16 bits, utilizando la instrucción DEC de 16 bits.

```
        LD    DE,1000
BUCLE   INC   BC
        LD    A,D
        OR    E
        JR    NZ,BUCLE
```

En este programa el registro BC aumenta 1.000 veces. Las desventajas que tiene esta técnica es que se usa un par de registros de 16 bits adicional, y el registro A interviene a la hora de ''comprobar el 0'' en el programa.

Obviamente, si queremos la opción STEP en el programa anterior, lo único que tenemos que hacer es añadir algunas instrucciones INC o DEC.

Por ejemplo:

```
        LD    C,0
        LD    B,100
BUCLE   INC   C
        INC   C
        DEC   B
        DJNZ  BUCLE
```

En el ejemplo anterior el registro C contaría: 0, 2, 4, ..., y el registro B lo haría de mayor a menor: 100, 98, 96, ... Ten cuidado de no asignar un número impar al registro B, ya que entonces este registro nunca llegaría al 0 y el bucle sería infinito.

La tabla siguiente te enseña algunas de las características de las instrucciones de salto y bucles que hemos visto hasta ahora. Los *flags* no son afectados por estas operaciones.

	Bytes	*Tiempo*
JP nn	3	10
JP cc,nn	3	10
JR d	2	12
JR C,d	2	7/12
JR NC,d	2	7/12
JR Z,d	2	7/12
JR NZ,d	2	7/12
JP (HL)	1	4
JP (IX)	2	8
JP (IY)	2	8
DJNZ d	2	8/13

Donde aparezcan dos tiempos, el primero se refiere al tiempo de la instrucción cuando NO se cumplió la condición, y el segundo es el tiempo que transcurrió cuando SI se cumplió la condición.

Las instrucciones CALL y RETURN

En BASIC tenemos las instrucciones GOSUB y RETURN, con las que podemos utilizar subrutinas. Una subrutina es un grupo de instrucciones que aparecen una vez en el programa pero que pueden ejecutarse tantas veces como queramos.

En el código máquina Z-80, podemos también usar esta misma posibilidad en nuestros programas; de hecho ya has usado una instrucción en código máquina equivalente a la GOSUB cuando ejecutaste la función USR para ejecutar tus programas en lenguaje máquina.

Dicha instrucción manda a la UCP que trate como una subrutina el trozo de lenguaje máquina que quieras; después lo ejecuta y vuelve al intérprete de BASIC cuando llega a la instrucción RET. En código máquina la instrucción

```
CALL nn
```

llama a la subrutina de la dirección *nn*. Como ves, la instrucción CALL ocupa tres bytes; un byte representa la instrucción y los otros dos son necesarios para especificar la dirección en la que empieza la subrutina.

Cualquier fragmento de programa que actúe como una subrutina tiene que terminar con la instrucción.

```
RET
```

que es la equivalente a la RETURN del BASIC. Una vez que se haya ejecutado la instrucción RET, la UCP seguirá ejecutando el programa a partir de la orden siguiente a la instrucción CALL; pero ¿cómo sabe la UCP dónde tiene que seguir ejecutando el programa?

Aquí es donde interviene la pila; siempre que se ejecuta una instrucción CALL, la UCP guarda en la pila la dirección de la instrucción que sigue a la instrucción CALL. Cuando se ejecuta la instrucción RET, el programa va a la pila y coge el último valor que haya en ella; este número es el que dejó la instrucción CALL y que se supone que es la dirección a la que debe volver el programa después de ejecutar la subrutina. La instrucción RET saltará a la dirección que le indique la pila en ese momento.

Dentro de una subrutina y siempre que utilices una pila, es absolutamente necesario que todas las instrucciones PUSH tengan su correspondiente instrucción POP para que así la instrucción RET pueda devolver a la UCP a la siguiente instrucción después de la llamada a la subrutina.

Hay algunos métodos con los que se modifica el valor de la pila al que la instrucción RET va a acceder, pero es mejor dejarlos aparte hasta que estés bien seguro de lo que estás haciendo; más adelante veremos algunos de estos métodos.

El funcionamiento de las instrucciones CALL y RET se ve muy bien en el siguiente diagrama:

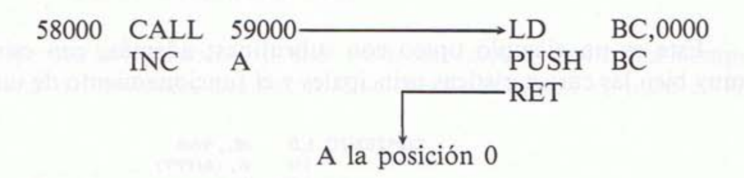
```
58000  CALL  59000 ──────────→ 59000  INC A
58003  INC   A ←────────────── 59001  RET
```

Si incluyes una instrucción PUSH y otra POP en esa subrutina tendrás lo siguiente:

```
58000  CALL  59000   59000 ──────→ PUSH  AF
       INC   A    ←──────┐         INC   A
                         │         POP   AF
                         └──────── RET
```

La instrucción PUSH asigna un número adicional en la pila que recobra la instrucción POP, con lo que la instrucción RET recibirá la instrucción correcta de la pila.

En el ejemplo siguiente, la pila no tiene el valor necesario que necesita la instrucción RET para devolver a la UCP a la posición adecuada, debido a que la instrucción PUSH no tiene su correspondiente POP.



En este ejemplo, la UCP saltará a la posición 0, que corresponde a la rutina RESET del MSX. De todas formas, en algunas aplicaciones es útil cambiar la dirección que debería encontrar la instrucción RET, pero NO es un método aconsejable de programación, y yo particularmente no se la aconsejaría a ningún principiante en el tema.

En el caso de que cambies el valor de la pila, usa la instrucción RET como una instrucción de salto; la rutina siguiente es un ejemplo de ello:

```
57000  CALL  59000   59000  POP   DE
                            LD    BC,58000
                            PUSH  BC
                            RET
```

La instrucción POP DE borra la dirección a la que volvería la UCP, es decir, la 570003. Después usamos la instrucción PUSH para poner otra dirección en la pila; al ejecutar la instrucción RET, hace que también se ejecute un salto. Como ves, es otra forma de usar la instrucción CALL, pero esta instrucción no fue pensada para desempeñar esa función, sino sencillamente para que llamara a una subrutina y fuera luego capaz de devolver la UCP a la instrucción que sigue a la CALL.

Fíjate que tienes que especificar la dirección completa de la rutina llamada; esto quiere decir que el programa que llama a las rutinas de la ROM del MSX son reubicables, pero, sin embargo, aquellos programas que llaman a las rutinas hechas por el usuario en la RAM no lo son.

Cómo guardar registros

Puede que quieras llamar a alguna subrutina y a la vez guardar el contenido de determinados pares de registros; esto lo puedes hacer usando la instrucción PUSH para meter pares de registros en la pila antes de ejecutar la instrucción CALL y luego usar la instrucción POP para sacar los pares de registros después de que la UCP haya ejecutado la subrutina. Obviamente, no es necesario que guardes todos los registros, sólo los que consideres necesarios, lo cual es particularmente importante si llamas a la subrutina con una interrupción. Si guardas los registros como antes te he explicado, se suele decir que han sido preservados o también que "los has protegido de la UCP".

Veamos ahora cómo usar una subrutina en el ejemplo siguiente.

EJEMPLO 19

Este es un ejemplo típico con subrutinas; además, con este caso aprenderás muy bien las características principales y el funcionamiento de una subrutina:

```
COMIENZO LD   HL,960
         IN   A,(&H99)
         XOR  A
         CALL SALIDA
         LD   A,64
         CALL SALIDA
BUCLE    LD   A,(&HF7F8)
         OUT  (&H98),A
         DEC  HL
         LD   A,L
         OR   H
         JR   NZ,BUCLE
         RET
SALIDA   OUT  (&H99),A
         RET
```

Hay varias cosas importantes a destacar del ejemplo anterior: lo primero es que cuando vayas a ensamblar el programa deja dos espacios para la dirección de la subrutina SALIDA en las instrucciones CALL.

Una vez que se conozca la dirección, puedes poner entonces los bytes en el programa y, como siempre, la dirección se almacena con el byte bajo primero. En segundo lugar, yo siempre suelo poner las definiciones de una subrutina de modo que no entorpezcan el programa principal, es decir, para que no puedan ejecutarse accidentalmente sin haberlas llamado.

Si eso ocurriera, puede que sucedan dos cosas: o bien que vuelvas al BASIC después de ejecutarse la instrucción RET del final de la subrutina, o bien que pierdas el control del sistema.

En tercer lugar, las llamadas a las subrutinas son siempre más lentas que cuando ejecutábamos el programa directamente, debido a que las instrucciones CALL y RET tardan un tiempo finito en ejecutarse. Es decir, aunque las subrutinas ahorran espacio en la memoria, tardan más tiempo en ejecutarse. Hay veces que ahorrar sitio en la memoria no es absolutamente necesario, pero sí lo es el tiempo

de ejecución. Es en esos casos donde repito partes de programa en vez de llamar a una misma subrutina.

Las instrucciones condicionales CALL y RETURN

La instrucción siguiente del BASIC:

IF ... THEN ... GOSUB nn

ejecuta la instrucción GOSUB siempre que se cumplan determinadas condiciones; pues bien, hay una instrucción muy similar a ésa pero en código máquina; me refiero a

CALL cc, dirección

que ejecuta la instrucción CALL en la dirección especificada siempre que se cumpla una determinada condición. Las condiciones que se usan en esta instrucción son:

CALL C	si hay acarreo
CALL NC	si no hay acarreo
CALL Z	si el resultado es 0
CALL NZ	si el resultado no es 0
CALL PE	si la paridad es par
CALL PO	si la paridad es impar
CALL M	si el signo del resultado es negativo
CALL P	si el signo del resultado es positivo

Ningún *flag* queda afectado por estas instrucciones CALL. Como has visto, podemos simular la instrucción ON ... GOSUB del BASIC con trozos de programas como los siguientes:

```
LD     A, (ELECCION)
CP     1
CALL   Z,OPCION 1
CP     2
CALL   Z,OPCION 2
```

etcétera. Fíjate en el hecho de que el registro A tienes que preservarlo antes de llamar a la subrutina o hacer esto como una de las primeras instrucciones de la subrutina y realmacenarla antes de dejar la subrutina; si no la vuelves a almacenar, es muy posible que el Z-80 escoja otra opción cuando vuelva de ejecutar una subrutina.

También hay instrucciones condicionales RET que sólo vuelven de una subrutina si se cumple cierta condición; dichas condiciones son las mismas que para la instrucción CALL.

Mnemónico	*Bytes*	*Tiempo*	C	Z	P/V	S	N	H
			Efectos en los flags					
Call dirección	3	17	—	—	—	—	—	—
Call cc,dirección	3	10/17	—	—	—	—	—	—
RET	1	10	—	—	—	—	—	—
RET cc	1	5/11	—	—	—	—	—	—

Notación para los flags:

- * indica que una operación ha afectado ese *flag*.
- 0 indica que el *flag* está a 0.
- 1 indica que el *flag* está a 1.
- — indica que no ha sido afectado ese *flag*.

La tabla anterior nos muestra algunas características de las instrucciones CALL y RET.

La instrucción RESTART

Esta instrucción es como una función CALL de un solo byte. El único problema que presenta esta instrucción es que sólo podemos llamar con ella a determinadas direcciones. Dichas direcciones tienen que estar en los primeros 256 bytes de la memoria del Z-80; por tanto, todas las instrucciones estarán dentro de la ROM del MSX.

La función de las instrucciones RESTART es realizar rápidas y efectivas llamadas a las subrutinas. Hay ocho instrucciones distintas RESTART, que nos permiten llamar a distintas direcciones. Las direcciones son: &H00, &H08, &H10, &H13, &H20, &H28, &H30 y &H38. Por ejemplo, la instrucción

```
RST &H00
```

ejecutaría un salto a la dirección 0, lo que provocaría que se ejecutara la instrucción RESET.

Interrupciones

Estas son especialmente útiles en un ordenador.

Una interrupción es una señal que se manda a la UCP para informarla de que se necesita su atención para solucionar cierta situación dentro del ordenador. La UCP anota lo que esté haciendo; luego salta a donde se la necesita y ejecuta el trabajo que le mande la interrupción. A esta rutina se la llama: RUTINA PARA EL TRATAMIENTO DE INTERRUPCIONES y normalmente guarda varios registros, ejecuta el trabajo necesario, realmacena los registros y vuelve a donde estaba con la instruccion

```
RETI
```

que es la abreviatura de *RETurn from Interrupt* (retorno de una interrupción). Dicha instrucción coge de la pila la dirección a la que tiene que volver de manera análoga a la que vimos con la instrucción RET.

En los MSX la forma más frecuente de interrupción se genera 50 veces por segundo por el VDP y la rutina para el tratamiento de interrupciones lee el teclado. La interrupción que genera el VDP provoca un salto a la dirección &H38.

Las interrupciones que has visto hasta ahora se suelen llamar ENMASCARABLES; hay otras formas de interrupciones en la UCP del Z-80 llamadas INTERRUPCIONES NO ENMASCARABLES; pero éstas no las voy a explicar en este libro, ya que no son muy útiles para el programador.

Una interrupción enmascarable puede ignorarse por la UCP si ésta ha sido programada para ello. La instrucción

```
DI
```

hace que la UCP ignore las interrupciones enmascarables. DI es la sigla de *Disable Interrupts,* que quiere decir desactivación de interrupciones.

La instrucción

```
EI
```

sigla de *Enable Interrupts,* que quiere decir activación de interrupciones, permite a la UCP responder a las interrupciones enmascarables.

Desactivando las interrupciones se aumenta ligeramente la velocidad de los programas; sin embargo, la UCP no preguntará nada al teclado si usas este tipo de interrupciones. En el capítulo siguiente veremos que da igual que no te pregunte nada; lo que sí es importante es que escribas la instrucción EI en tus interrupciones antes de volver al BASIC, porque, si no, el teclado seguiría bloqueado, aun cuando volvieras al BASIC, y, como es lógico, la capacidad de tu ordenador se vería mermada en gran parte. Imagínate lo que ocurriría si no pudieras usar el teclado de tu ordenador.

Aunque vayamos a estudiar las interrupciones en el capítulo 10, éste no es realmente un tema para los que seáis principiantes en este lenguaje máquina, pero, si algunos de vosotros tenéis interés en este aspecto de la programación y no quedáis satisfechos con las explicaciones del capítulo 10, os aconsejo que busquéis algún otro libro más avanzado sobre este lenguaje máquina.

Con este último epígrafe se completan las instrucciones que controlan programas; ahora vamos a ver otro tipo de operaciones importantes dentro de la programación del Z-80 que usan saltos y bucles para funcionar automáticamente con más de un byte; me refiero a las operaciones en bloque.

Operaciones de bloques

Estas instrucciones suelen actuar normalmente sobre más de uno o dos bytes, como has visto que hacían las otras instrucciones hasta ahora. La instrucción más sencilla que actúa sólo sobre un byte dentro de estas instrucciones es la instrucción

CPI

que es la abreviatura de *ComPare with Increment* (comparación con incremento). Esta instrucción compara el contenido del registro A con el (HL), y automáticamente se incrementa el par HL. Así, después de que se ejecute esta instrucción, el par HL señalará el byte siguiente para hacer otra comparación.

Normalmente, esta instrucción sirve para buscar en la memoria un determinado byte; una vez que se encuentra este byte, se activará el *flag* Z. El ejemplo 20 es una muestra de lo que hace esta instrucción.

EJEMPLO 20

```
        LD   A,(&HF7F8)
        LD   HL,0000
BUSCA   CPI
        JR   NZ,BUSCA
        DEC  HL
        LD   (&HF7F8),HL
        LD   A,2
        LD   (&HF663),A
        RET
```

Introduce en el ordenador el programa anterior; la posición para la instrucción JRNZ es —4 o FD, ya que la instrucción CPI ocupa dos bytes.

Dicho programa busca un determinado byte de la memoria a partir de la posición 0. El byte que vamos a buscar se pasa con la instrucción USR como un parámetro entero y se le asigna al acumulador de la posición &H7H8. Para ver esto en la práctica, haz que la instrucción DEFUSR señale a la primera instrucción del programa y después teclea:

PRINT USR(101)

dando por hecho que ya has usado la instrucción DEFUSR. La rutina devolvería el valor 3519, que es la primera posición de la ROM de tu MSX que tiene el valor 101. Fíjate si va rápido este programa en lenguaje máquina, que, mientras el ordenador ha vuelto al BASIC, él ya ha buscado un byte en 3519 direcciones.

La instrucción CPD desempeña una función muy parecida a la instrucción anterior, pero con la diferencia de que ésta disminuye el registro HL en vez de incrementarlo. Ambas instrucciones, la CPI y la CPD, decrementan el par de registros BC, lo cual es muy útil, ya que nos permite buscar los bytes en zonas de memoria de determinada longitud.

El principio de la zona en la que vamos a buscar dichos bytes está en el registro HL, y el número de bytes de la zona que tenemos que examinar está en el

registro BC. El ejemplo 21 es un caso práctico de lo que te acabo de explicar y se encarga de buscar un byte entre 255 bytes.

EJEMPLO 21

```
        LD   A,(&HF7F8)
        LD   BC,255
        LD   HL,0000
BUSCA   CPI
        JR   Z,SALIDA
        INC  C
        DEC  C
        JR   NZ,BUSCA
SALIDA  LD   (&HF7F8),HL
        LD   A,2
        LD   (&HF663),A
        RET
```

Lo más destacable del programa anterior es que estamos examinando el registro menor del par de registros BC, es decir, el registro C. Lo que hice fue escribir una instrucción INC seguida de una DEC, que nos permitirá saber si es 0 el registro C. La instrucción DEC activa el *flag* Z para que puedas saber si en el registro C hay un 0.

Este programa devuelve el valor 255 si no ha encontrado el byte buscado en los primeros 255 bytes de la memoria; si, por el contrario, lo encuentra, el programa nos devolverá la posición de dicho byte.

Las instrucciones CPIR y CPDR

Estas dos instrucciones son de gran utilidad, ocupan dos bytes cada una y son el equivalente de las instrucciones CPI y CPD pero con un salto.

La UCP busca automáticamente en una zona de memoria hasta que encuentre bien el byte o bien el fin de la zona de memoria. El registro A especifica el byte que se va a buscar; el par de registros HL tiene el principio de la zona que queremos buscar y el registro BC el número de bytes que tiene que examinar. La instrucción se acaba por una de las dos causas siguientes:

1. Porque se encuentre el byte buscado.
2. Porque se ha llegado al final de la zona.

Como es lógico, después de ejecutarse una instrucción CPIR, tendremos que saber cuál de las dos causas ha determinado que se termine de ejecutar la instrucción CPIR.

```
LD      HL,0000
LD      BC,&HFFFF
LD      A,255
CPIR
LD      A,B
OR      C
JR      Z,END
```

Pues bien, si cuando se termina de ejecutar la instrucción anterior el resultado del registro BC es 0, se ejecutará un salto a la etiqueta "END", lo cual indicaría que ya ha sido examinado todo el bloque y no se ha encontrado el byte buscado. Si no es así, se deberá a que la instrucción se ha parado porque ha encontrado el byte.

Para que te hagas una idea del tiempo que tarda en ejecutarse dicha instrucción, te diré que, por ejemplo, este último programa tardó 0.12 segundos en buscar el byte 101 entre 12.990; en los programas en código máquina esto supone, aunque te parezca increíble, mucho tiempo. Aunque usando las instrucciones CPIR y CPDR o las instrucciones CPI y la CPD sea más rápido que usar rutinas escritas con las instrucciones separadas CP y JR.

Movimientos de bloques

Hay veces que necesitarás mover fragmentos de memoria dentro del ordenador. Esto lo puedes hacer con el siguiente programa:

```
        LD      HL,0000
        LD      B,200
        LD      DE,58000
BUCLE   LD      A,(HL)
        LD      (DE),A
        INC     HL
        INC     DE
        DJNZ    BUCLE
```

Donde se transfieren 200 bytes de las direcciones de la 0 a la 200, a las direcciones 58000 hasta la 58200. El par de registros HL se usa para señalar el byte que estemos copiando y el DE para determinar adónde vamos a copiar dicho byte. El registro B lo usaremos para contar el número de bytes transferidos.

Aunque el programa anterior funciona no es la forma más apropiada para mover bloques de memoria en el ordenador, ya que el lenguaje máquina Z-80 tiene instrucciones que harían transferencias de bytes directamente. La primera instrucción que vamos a ver de este tipo es la instrucción LDI.

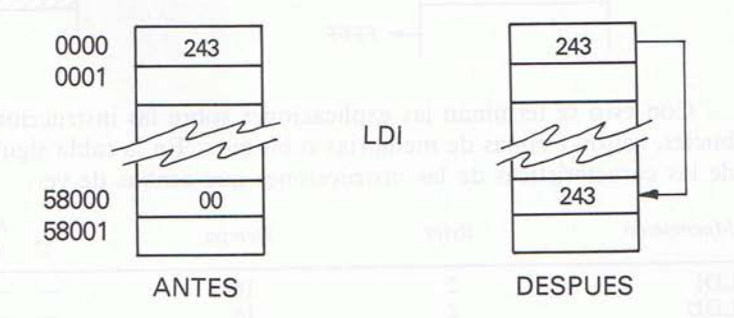
```
        LD      HL,0000
        LD      DE,58000
        LD      BC,100
BUCLE   LDI
        LD      A,B
        OR      C
        JR      NZ,BUCLE
        RET
```

Este programa transfiere los 100 bytes que hay desde la dirección 0 en adelante, a la posición 58000 hasta la 58100. El registro DE se suele denominar de DESTINO, ya que tiene la primera dirección en donde van a transferirse o copiarse los

bytes, y al registro HL se le llama registro FUENTE, ya que tiene la dirección del primer byte que quieres copiar.

Una vez que se haya ejecutado la instrucción LDI, los registros HL y DE se incrementarán y el registro BC se decrementará. La instrucción LDD provoca algo parecido a esto, pero en este caso los pares de registros HL y DE más bien disminuyen que aumentan.

Fíjate en el diagrama siguiente.



Después de ejecutarse la instrucción LDI, el par HL indicará la posición 0001 y el par de registros DE la posición 58001. Después de ejecutar las instrucciones LDI y LDD, y si el par de registros BC no es igual a 0, el *flag* P/V se activará. Por tanto, podemos usar dicho *flag* para saber si necesitamos repetir la instrucción o no.

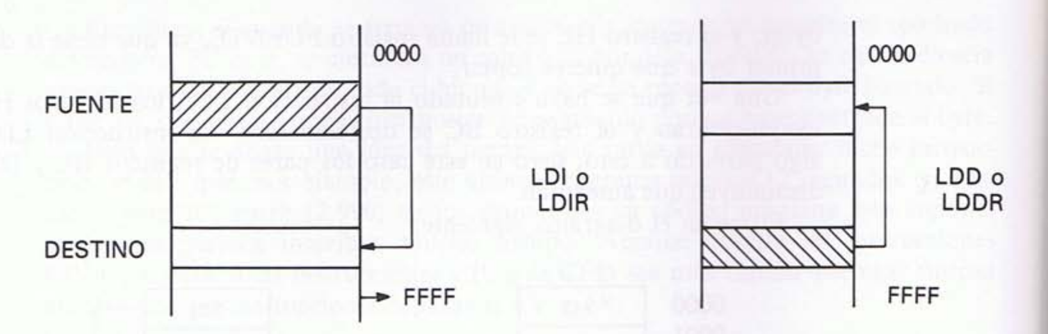
Obviamente, este método es más eficiente que el primero que vimos. Sin embargo, hay aún otro método más eficiente para repetir las instrucciones LDI o LDD; me refiero a las instrucciones

LDIR o LDDR

que examinan automáticamente el *flag* y repiten el proceso anterior si es necesario. Por ejemplo, el programa

```
LD    HL,0000
LD    BC,1000
LD    DE,58000
LDIR
RET
```

transfiere automáticamente 1.000 bytes desde la dirección 0 a la 1000 a la dirección 58000 en adelante. La elección entre las instrucciones LDIR y LDDR es muy fácil de tomar.



Con esto se terminan las explicaciones sobre las instrucciones que operan con bucles, saltos y zonas de memorias o bloques. En la tabla siguiente tienes algunas de las características de las instrucciones que acabas de ver.

Mnemónico	*Bytes*	*Tiempo*	*Efectos en los* flags C	Z	P/V	S	N	H
LDI	2	16	—	—	*	—	0	0
LDD	2	16	—	—	*	—	0	0
LDIR	2	21/16	—	—	0	—	0	0
LDDR	2	21/16	—	—	0	—	0	0
CPI	2	16	—	*	*	*	1	*
CPD	2	16	—	*	*	*	1	*
CPIR	2	21/16	—	*	*	*	1	*
CPDR	2	21/16	—	*	*	*	1	*

Notación para los flags:

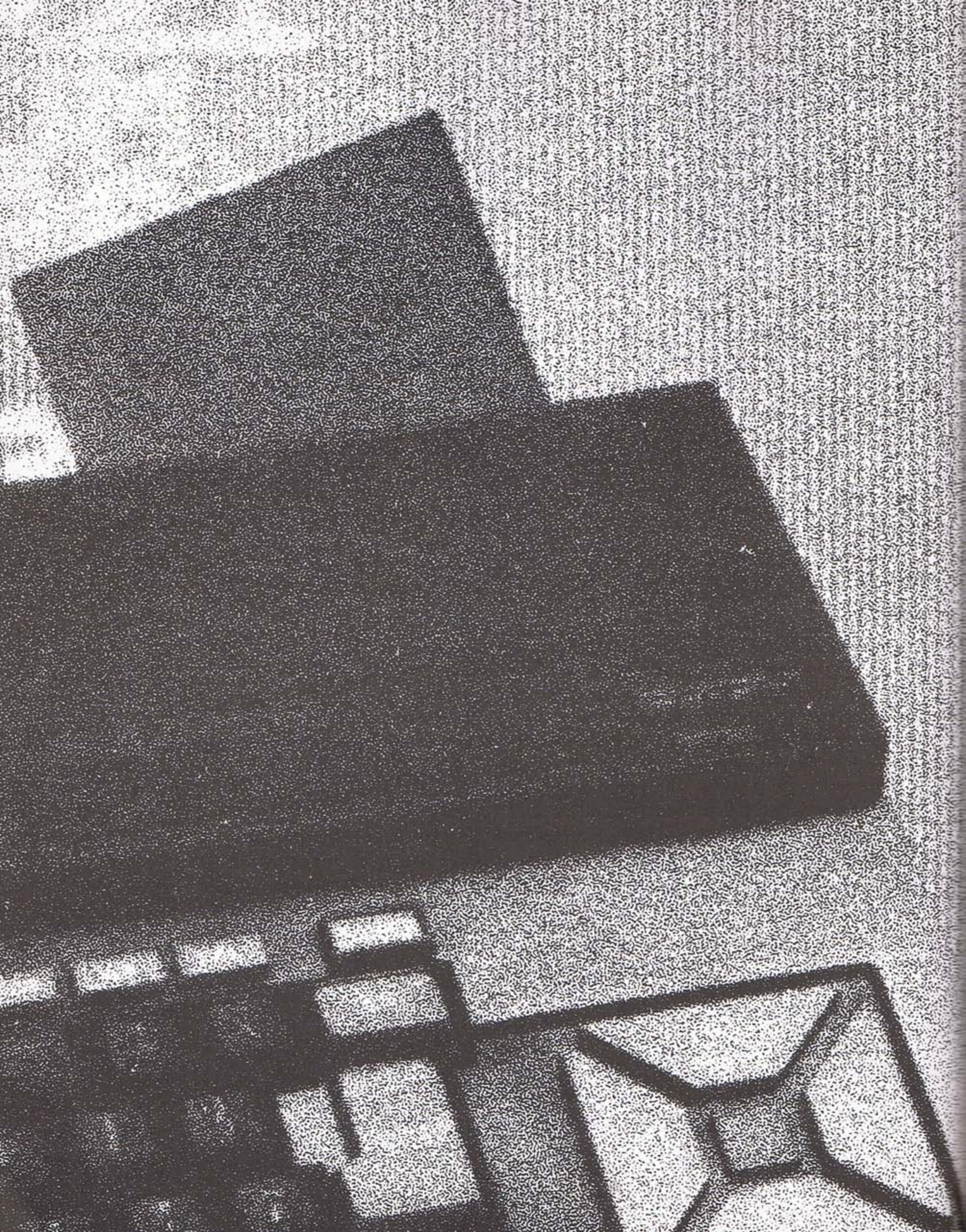
* indica que una operación ha afectado ese *flag*.
0 indica que el *flag* está a 0.
1 indica que el *flag* está a 1.
— indica que no ha sido afectado ese *flag*.

Se puede decir, sin temor a equivocarnos, y si has asimilado lo que hasta aquí he explicado, que ya eres capaz de crear programas en código máquina que ejecuten la mayoría de operaciones que los programas en BASIC y que el MSX es capaz de ejecutar. Sin embargo, hay tres cosas que todavía no sabes hacer en código máquina, a saber:

- Leer el teclado.
- Generar sonidos.
- Escribir textos o dibujar gráficos en la pantalla (la más importante de todas).

Las instrucciones del Z-80 que realizan estas funciones, así como las que controlan el cassette y el mando de juegos, son las instrucciones de entrada/salida (IN/OUT), que son las que vamos a ver en el siguiente capítulo. Una vez que sepas algo más sobre dichas instrucciones, veremos cómo manejar el VDP y el PSG de los ordenadores MSX.





9

Instrucciones de entrada y salida

Ya hemos visto cómo la UCP del Z-80 direcciona hasta 65536 posiciones de memoria diferentes. Pues bien, además podemos direccionar 256 posiciones distintas conocidas como direcciones de ENTRADA/SALIDA.

Dichas direcciones no están en la RAM, sino que son las direcciones del VDP, PPI y del PSG, en las que la UCP puede escribir. Cada uno de los dispositivos anteriores están dispuestos en direcciones diferentes de entrada/salida, de forma parecida a como lo están la ROM y la RAM en la memoria del MSX para poder acceder a ellas. Estos dispositivos se dice que están ''DIRECCIONADOS'' en esas posiciones de memoria.

Empecemos viendo las instrucciones que hacen posible las operaciones que he mencionado un poco antes; me refiero a las instrucciones IN y OUT. Las más sencillas son las siguientes:

```
IN     A,(N)
OUT    (N),A
```

donde *N* es un número entre 0 y 255.

La instrucción IN copia el valor del dispositivo direccionado como puerto *N*. También es posible tener dos dispositivos diferentes direccionados en una sola dirección de entrada/salida: un dispositivo se lee con la instrucción IN y al otro se le manda información con la instrucción OUT.

La instrucción OUT copia el contenido del registro A al dispositivo direccionado en la posición *N*. Por ejemplo, las instrucciones

```
LD   A,8
OUT  (&HA0),A
LD   A,15
OUT  (&HA1),A
```

formarían parte de un programa MSX y escriben el valor 15 del registro 8 en el generador de sonido programable. En el capítulo 11 veremos más detalles sobre cómo escribir en el dispositivo anterior; de momento sólo te diré que el equivalente a las instrucciones en código máquina anteriores es

```
SOUND 8,15
```

Algunas direcciones de la zona de entrada/salida del MSX no se usan y otras el usuario no necesita acceder a ellas, a menos que esté muy seguro de lo que está haciendo. Por tanto, no te voy a dar una descripción detallada de la zona de entrada/salida del MSX; solamente decirte las direcciones que necesitarás conocer sobre el procesador de video, el generador de sonido programable y el interfaz programable de periféricos.

Las instrucciones IN y OUT que hemos visto anteriormente son instrucciones que ocupan 2 bytes: uno tiene el código de operación, y el segundo, la dirección de entrada/salida. Las únicas direcciones que nos interesan son las siguientes:

&HA8 - &HAB

Son las direcciones que usamos para comunicarnos con el interfaz programable de periféricos. Este proceso de comunicación lo explicaré más adelante en este mismo capítulo.

&HA0 - &HA1

Estas direcciones nos permiten acceder al generador programable de sonido, que será explicado en el capítulo 11.

&H98 - &H99

Las direcciones &H98 y &H99 controlan el procesador de video (VDP, mira el capítulo 10). Estas direcciones son las mismas en todos los ordenadores MSX y nos permiten acceder a los registros de los dispositivos antes mencionados. Puedes controlar el contenido de dichos registros internos simplemente escribiendo o leyendo las direcciones de entrada y salida, con lo que controlas el contenido y el funcionamiento de esos registros.

Operaciones de entrada/salida con el registro C

Hay instrucciones que usan el registro C para guardar la dirección del dispositivo de entrada/salida al que quieres acceder. Estas instrucciones son las siguientes:

```
IN   r,(C)
OUT  (C),r
```

donde *r* es un registro de 8 bits. De esta manera, ahora podríamos escribir el programa anterior para escribir en el PSG, de la siguiente forma:

```
LD   C,&HA0
LD   A,8
OUT  (C),A
INC  C
LD   A,15
OUT  (C),A
RET
```

usando el registro C.

La orden IN siguiente

```
IN A,(C)
```

actúa sobre los *flags* de signo, cero y de paridad dependiendo de los datos leídos en el dispositivo en cuestión. Como ves, es posible examinar los *flags* después de haberse ejecutado esta instrucción.

Instrucciones de entrada/salida de bloques

Ya vimos las instrucciones para transferir zonas de memoria, LDI y LDIR, que transferían datos de unas posiciones de memoria a otras. Las que ahora te voy a explicar son menos útiles y nos permiten transferir datos desde posiciones de memoria determinadas a un dispositivo de la zona de entrada/salida.

Como ya te he dicho, estas instrucciones son más limitadas o restringidas a la hora de usarlas, debido a que sólo pueden transferir hasta 256 bytes de la memoria a un dispositivo de entrada/salida.

Las instrucciones INI e INIR

La instrucción INI transfiere un byte de datos de un dispositivo de entrada/salida a la memoria. En el registro C debe estar la dirección del dispositivo de entrada/salida; la instrucción INI sustituye a las instrucciones siguientes:

```
IN   A,(C)
LD   (HL),A
INC  HL
```

Por tanto, antes de ejecutar una instrucción INI, el par de registros HL tiene que tener la dirección de la posición de memoria a la cual quieres escribir el byte leído del dispositivo de entrada/salida.

Después de ejecutar la instrucción INI, el registro HL señalará la posición de memoria siguiente y el registro B se disminuirá. La instrucción INIR no es más que una instrucción INI pero con función iterativa. El programa siguiente transfiere 200 bytes de datos de un dispositivo de entrada/salida cuya dirección está en el registro C, a otros 200 bytes de memoria determinados por el par de registros HL.

```
LD   B,200
LD   HL,58000
LD   C,DIRECCION
INIR
RET
```

Los bytes leídos en el dispositivo ocuparían en la memoria desde la posición 58000 a la 58199. Si estamos limitados a transferir datos de un dispositivo de entrada/salida a la memoria es debido a que usamos el registro B como contador.

La función de las instrucciones IND e INDR es similar a las anteriores, con la diferencia de que el contenido del par de registros HL se va disminuyendo al final de cada transferencia en vez de incrementarse.

Las instrucciones OUT y OUTI

Estas instrucciones ejecutan la misma función que la INI y la INIR, pero a la inversa. El par de registros HL se usa para apuntar un byte de memoria que luego se pasará al dispositivo de entrada/salida. La dirección de ese dispositivo estará en el registro C; por ejemplo:

```
LD   HL,58000
LD   B,200
LD   C,DIRECCION
OTIR
RET
```

sacará 200 bytes a través del dispositivo cuya dirección esté en el registro C. El primer byte que se mandará fuera será el de la posición 58001; el segundo byte, el de la posición 58002; el tercer byte, el de la 58003, etc.

Con esta última explicación hemos terminado de ver el conjunto de instrucciones Z-80. El resto del libro lo voy a dedicar a explicarte cómo podrás usar las instrucciones que ya sabes para manejar los dispositivos del MSX.

Sólo queda por nombrar una instrucción que pudiera serte necesaria, pero lo divertido de esta instrucción es que no hace absolutamente NADA; me refiero a la instrucción NOP y, como bien supones, lo único que hace es desperdiciar tiempo. Te será muy útil sobre todo cuando necesitas realizar retardar el tiempo en tus programas.

El interfaz programable de periféricos

Este dispositivo, conocido también como PPI porque responde a las iniciales *Programmable Peripheral Interface,* es el responsable de algunas funciones de ENTRADA y SALIDA del ordenador MSX y lo estudiaremos en este capítulo ya que está dedicado a las instrucciones de entrada/salida.

El dispositivo PPI tiene cuatro registros internos y puedes acceder a ellos leyendo o escribiendo en determinadas direcciones de la zona de entrada/salida. Uno de esos registros es de control y se le denomina REGISTRO DE SELECCION DE MODO; el resto de los registros son registros de entrada/salida. El Z-80 toma como dispositivos de entrada o salida los registros anteriores, dependiendo de cómo se activa el registro de selección de modo. En general, se puede decir que no es aconsejable cambiar el contenido del registro de selección de modo, porque, si lo cambias, pueden ocurrir algunos pequeños desastres que van desde confundir el teclado hasta provocar que el ordenador ignore algunas partes de la memoria e incluso la memoria entera.

Cuando enciendes tu ordenador, el sistema operativo del MSX activa el registro de selección de modo. La tabla siguiente muestra cómo se configuran los otros tres registros en condiciones normales.

Número del registro	*Dirección*	*IN/OUT*
A	&HA8	OUTPUT
B	&HA9	INPUT
C	&HAA	OUTPUT

Pero, ¿qué hacen todos estos registros exactamente?

Pasemos a verlo.

El registro A

La función de este registro se sale un poco de lo que pretende este libro; sin embargo, te diré que controla la disposición de la memoria dentro del MSX.

Para saber más acerca de este registro, consulta el libro *Descubre tu MSX,* ya que considero que no es un tema para principiantes y, como ya te he dicho, no lo explicaré en este libro.

El registro B

Este registro de entrada se usa para leer el teclado junto con los 4 bits bajos del registro C.

El registro C

Es un registro de salida muy útil; los 4 bits bajos de este registro ayudan a leer el teclado. Este registro lo explicaré más adelante en este mismo capítulo.

Veamos ahora los otros bits del registro:

El bit 4

Este es la señal de control del cassette y activa el relé de control remoto del motor del cassette. Cuando este bit se pone a 0, se desactiva el relé. Si, como el mío, tu ordenador MSX hace un ruido cuando se ejecuta un relé, oirás también una especie de ''clak'' cuando el bit se active y desactive. Cuando se cierra este relé y el enchufe se conecta al conector apropiado del cassette, éste empezará a funcionar cuando pulses PLAY o RECORD. Si está abierto el relé, el motor del cassetteno funcionará.

El bit 5

Este bit está relacionado con la transferencia de datos entre el ordenador MSX y el cassette.

El bit 6

Controla el estado de la tecla CAPS LOCK, que tiene una luz y está en el teclado de tu ordenador. Cuando este bit está a 0, la luz se encenderá, y cuando el bit se activa, la luz se apagará. El estado del bit no afecta al estado de la función CAPS LOCK, sólo al estado de la luz.

El bit 7

Cuando este bit pasa de 1 a 0, oirás un ''click'' dentro del ordenador. Como verás en el ejemplo siguiente, podemos usar este bit para generar un sonido. Ten

en cuenta que, si generas sonido, de este modo estarás realizando un proceso intensivo. En ese caso conviene que sepas que sería la UCP del Z-80 la que tendría que hacer todo el trabajo, mientras que, si utilizas el dispositivo que tiene el MSX para generar sonido, la UCP no tendrá casi que intervenir en el proceso y podrás utilizarla para otras cosas, como te explicaré en el capítulo 11.

EJEMPLO 22

En este ejemplo, usaremos el bit 7 del registro C del PPI para generar sonidos:

```
        LD   HL,&H0100
BUCLE   LD   A,255
        OUT  (&HAA),A
        LD   B,255
BUCLE1  DJNZ BUCLE1
        LD   A,127
        OUT  (&HAA),A
        LD   B,255
BUCLE2  DJNZ BUCLE2
        DEC  HL
        LD   A,H
        OR   L
        JR   NZ,BUCLE
        RET
```

El ruido se genera asignando unos y ceros al bit correspondiente. La duración del sonido depende del contenido del par de registros HL. Este registro tiene el número de ciclos de unos y ceros que se repetirán para generar sonido. La duración del sonido se determina con el contenido del registro B, antes de los dos bucles de retardo: BUCLE1 Y BUCLE2.

Si introduces este programa en el ordenador y lo ejecutas, te darás cuenta de que el sonido que oirás no es siempre igual; esto se debe a que el VDP está todavía generando interrupciones cuando se está ejecutando la rutina anterior y, por tanto, el tono generado será interrumpido por el dispositivo anterior.

Para conseguir un sonido homogéneo: desactiva dichas interrupciones que interceptan la rutina de generar sonido y hazlas funcionar cuando vuelvas al BASIC. Para hacer esto, no tienes más que poner una instrucción DI inmediatamente después de la LD HL y una instrucción EI antes del último RET. También es posible variar la duración del sonido poniendo instrucciones NOP en los BUCLE1 y BUCLE2; no te olvides de modificar los saltos relativos del programa si haces esto último.

Los bits 0 a 3

Estos bits se usan, como ya he dicho, para leer el teclado del ordenador. Ahora vamos a ver cómo se utilizan las instrucciones IN y OUT para detectar cuándo pulsas ciertas teclas en programas en código máquina. Aunque voy a darte una explicación bastante amplia sobre cómo se puede leer el teclado del MSX, no te voy a

facilitar, por ejemplo, una rutina que te permita leer cadenas alfanuméricas desde un programa en código máquina.

Estos cuatro bits juntos forman lo que se llama la señal para examinar el teclado del MSX. Cuando el ordenador lee el teclado, un valor de 4 bits se obtiene de esos cuatro bits y se lee el registro B del PPI. El valor leído de este registro y los cuatro bits que se llevaron al registro C nos permiten detectar pulsaciones de teclas. Si no te ha que dado esto último muy claro, vete al plano del teclado del MSX.

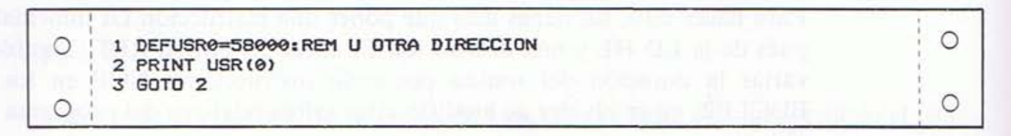
Por ejemplo, cuando ponemos el número binario 1000 en los cuatro bits bajos del registro C, el valor releído del registro B dependerá de si pulsas las teclas que mueven el cursor, o la de la barra espaciadora, la de borrar, la de insertar o las HOME/CLS. En el caso de que pulses la barra espaciadora, el bit 0 del registro C se pondría a 0, mientras que los bits que representan las otras estarían a 1 si pulsas varias teclas a la vez, los bits que representan a cada una de esas teclas se pondrán a 0. El ejemplo 23 es un caso práctico de cómo se lee el teclado.

EJEMPLO 23

En esta rutina, veremos lo que ocurre cuando pulsas las teclas que mueven el cursor, la barra espaciadora, la de borrar y la de insertar y las teclas HOME/CLS. He elegido estas teclas, porque son las que normalmente se usan en programas de juegos, como dispositivos alternativos al mando de juegos. Pasemos a ver la rutina en cuestión:

```
LD   A,&HF8
OUT  (&HAA),A
IN   A,(&HA9)
LD   (&HF7F8),A
LD   A,2
LD   (&HF663),A
XOR  A
LD   (&HF7F9),A
RET
```

Una vez que hayas metido esta rutina en tu ordenador, comprenderás mejor lo que hace el programa siguiente en BASIC:



```
1 DEFUSR0=58000:REM U OTRA DIRECCION
2 PRINT USR(0)
3 GOTO 2
```

Ejecútalo y pulsa cualquiera de las teclas mencionadas anteriormente. Fíjate que el número devuelto por la rutina en código máquina es diferente al pulsar teclas distintas. Para saber qué teclas has pulsado, por ejemplo en un programa de juegos, necesitaríamos conocer los distintos bits del byte, para lo cual simplemente leeríamos otra vez el registro B del PPI y utilizaríamos la instrucción BIT para determinarlo.

Si quieres modificar la rutina anterior para que sea capaz de reconocer otras teclas además de las que te indico, cambia el valor mandado al registro C del PPI.

PLANO DEL TECLADO DEL MSX

REGISTRO B

REGISTRO C (4 bits menos significativos)	MSB 7	6	5	4	3	2	1	LSB 0
0000	7	6	5	4	3	2	1	0
0001	;	[	@	¥	∧	−	9	8
0010	B	A	—	/	.	,	]	:
0011	J	I	H	G	F	E	D	C
0100	R	Q	P	O	N	M	L	K
0101	Z	Y	X	W	V	U	T	S
0110	F3	F2	F1	かな	CAPS	GRAPH	CTRL	SHIFT
0111	RETURN	SELECT	BS	STOP	TAB	ESC	F5	F4
1000	→	↓	↑	←	DEL	INS	HOME CLS	SPACE

Fíjate que yo he mandado el valor &HF8 al registro C; puede que te preguntes: ¿por qué no sencillamente el 08? Pues bien, decidí dejar los 4 bits más altos del registro C a 1 mientras se estuviera ejecutando dicha rutina, para prevenir posibles sonidos generados por la ejecución del programa anterior y, además, para prevenir que el cassette funcionase.

Cómo leer el teclado completo

Los principios fundamentales para leer el teclado son los mismos que hemos visto anteriormente; simplemente tienes que poner un determinado valor de 4 bits en los 4 bits bajos del registro C y después leer el registro B para conseguir el valor correspondiente. Para examinar el teclado completo, el valor de 4 bits se incrementaría desde 0 a 8, hasta que todas las filas del teclado hayan sido examinadas o hasta que en el registro B haya otro valor distinto al 255.

Esta última condición indica que se ha pulsado un tecla. Además, podrías usar alguna tabla en la memoria que devolviera el código ASCII de la tecla pulsada. Sin embargo, sólo necesitamos mirar determinadas teclas pulsadas en un programa.

Fíjate en el ejemplo 24, que te enseñará a escribir un programa en código máquina que espere hasta que se pulse una tecla.

Todas las rutinas de lectura del teclado que hemos visto en este capítulo funcionan cuando se desactivan las interrupciones.

EJEMPLO 24

Como ya te he dicho, esta rutina hace que un programa se pare hasta que pulses una tecla, con la excepción de la tecla RESET. Puedes modificarlo de modo que espere a que pulses una determinada tecla.

```
BUCLE  LD   A,&HF0
BUCLE1 LD   (59000),A
       OUT  (&HAA),A
       IN   A,(&HA9)
       CP   &HFF
       RET  NZ
       LD   A,(59000)
       INC  A
       CP   &HF9
       JR   Z,BUCLE
       JR   BUCLE1
```

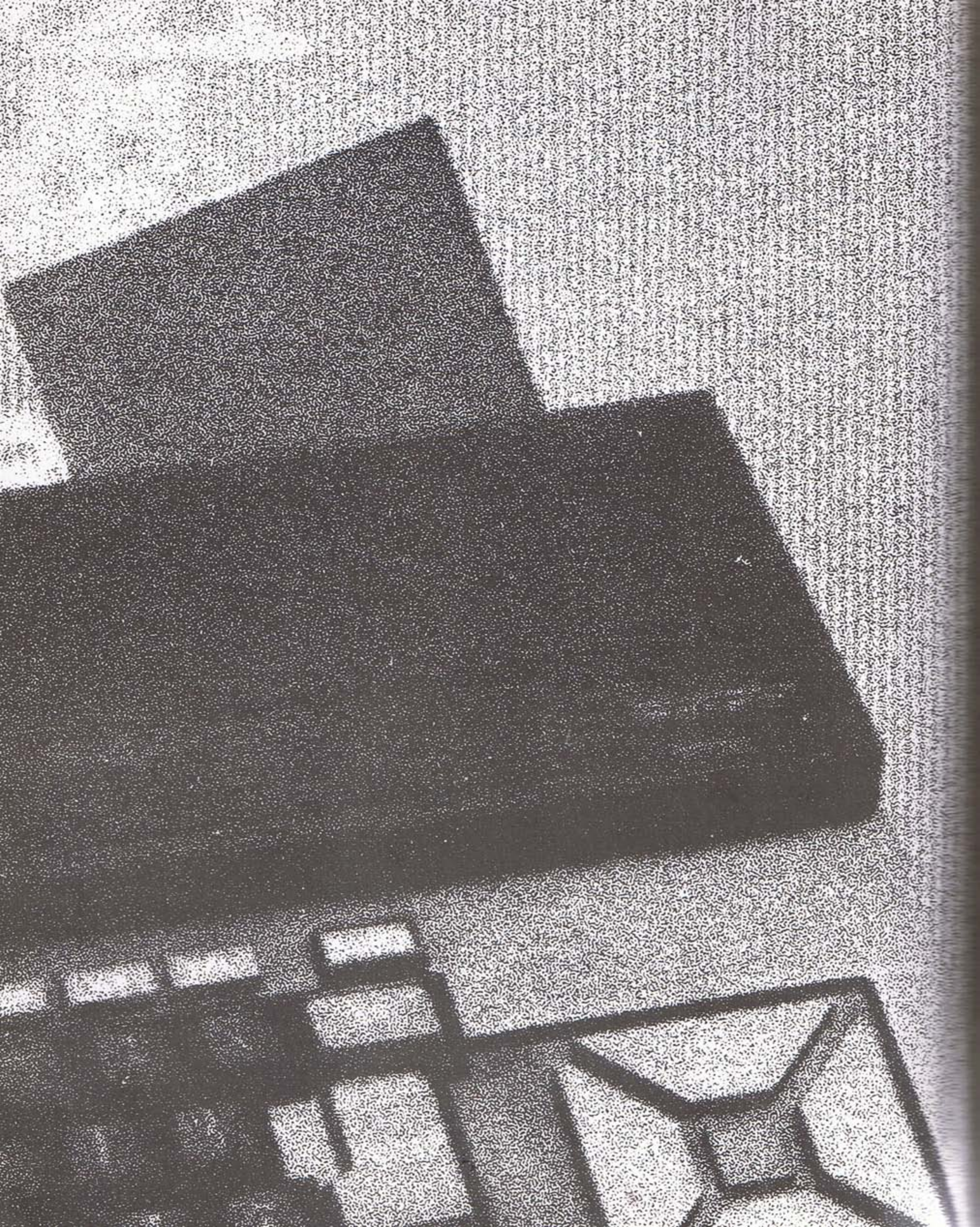
Escribe los bytes que forman este programa a partir de la dirección 58000. Este código simplemente espera hasta que el valor devuelto del registro B no sea 255.

Como puedes ver, si usas el código máquina para leer el teclado, tienes algunas ventajas:

1. Podemos detectar si pulsas varias teclas a la vez.
2. Es más rápido.
3. Podemos saber si has pulsado teclas como la CTRL o la SHIFT.

Como desventaja podemos señalar que es un poco más complicado conseguir códigos ASCII de los valores leídos a través del registro B; pero, si lo piensas un poco, ya casi has introducido cadenas alfanuméricas como haces normalmente en BASIC.

Ahora que ya conoces el funcionamiento de las instrucciones IN y OUT y las de manejo de bloques equivalentes, podemos pasar a usar el VDP y el PSG con programas en código máquina. Para aquellos de vosotros que no estáis familiarizados aún con el manejo de estos dispositivos desde el BASIC, os facilito una descripción de cómo se programan cada uno de estos potentes dispositivos y luego pasaré a explicarte algunos ejemplos prácticos de lo que hacen cada uno de estos dispositivos.



10 El procesador de video (VDP)

Este capítulo está dedicado a la programación del VDP, componente fundamental de los ordenadores MSX. Estudiaremos cómo escribir textos y gráficos en la pantalla, así como manejar SPRITES en código máquina.

A pesar de la complejidad del VDP, accedemos a él con sólo dos posiciones de la zona de entrada/salida del MSX.

El VDP

El VDP es un dispositivo también llamado TMS9918, y tiene acceso a 16384 bytes de la RAM de video. La UCP sólo puede acceder a la VRAM usando el VDP.

Hay cuatro tipos principales de transferencias de información entre la UCP y el VDP.

1. La UCP envía bytes a los registros del VDP.
2. La UCP envía bytes a la VRAM.
3. La UCP lee bytes del registro de estado del VDP.
4. La UCP lee bytes de la VRAM.

Puedes acceder al VDP a través de las posiciones de entrada/salida &H98 y &H99. Usando dichas direcciones, podemos escribir información en los registros del VDP y de la VRAM. En el próximo capítulo veremos ejemplos prácticos de esto último.

Los registros del VDP

Se puede decir que un registro del VDP es como un byte de la RAM que controla las operaciones del VDP. Ten en cuenta que NO es un byte de la VRAM. Dentro del VDP hay ocho registros que se denominan "sólo de escritura"; dichos registros no pueden mandar datos a la UCP. Sólo hay un registro "sólo de lectura" en el VDP, también llamado registro de estado; es como si fuera el registro *flag* del VDP. Los registros sólo de lectura se activan cuando enciendes la ROM del MSX. Si alteras siempre el contenido de los registros del VDP en código máquina, debes anotar lo que haya escrito, porque, si no, no podrías volver a leer ese registro.

De hecho, el programa en las ROM del MSX guarda una copia del valor del registro VDP en la zona de trabajo del sistema para que vigile lo que ocurre. A los valores que se asigna al VDP cuando dicho periférico se enciende, se accede con la instrucción BASE del BASIC, y las copias de la RAM se guardan, de manera que podamos modificar el contenido del VDP y leer el contenido del registro con la instrucción VDP().

Pero, ¿qué clase de información contienen los registros VDP? Pues bien: algunos registros tienen información que sirve al VDP para saber cómo está dipuesta la VRAM; otros registros tienen series de bits y cada uno de ellos controla algunos aspectos de las operaciones del VDP. Debido a esto, no sería una buena idea meter valores aleatorios en los registros VDP.

Numeración en los registros VDP

Los bits del byte de un registro VDP están numerados como sigue:

7	6	5	4	3	2	1	0

La RAM de video

Las 16K de la VRAM se disponen de forma diferente en los distintos modos de pantalla, como veremos más adelante. A la VRAM, en BASIC, accedemos con las instrucciones VPOKE y VPEEK; en código máquina accedemos a través de las dos posiciones de entrada/salida antes mencionadas.

Cómo acceder a los registros

En BASIC, solemos escribir en los registros con la instrucción:

VDP(n) = variable

donde *n* es el número de un registro. En código máquina usaremos la instrucción OUT para escribir datos en los registros VDP. La operación de escribir un valor en un registro del VDP se compone de dos instrucciones OUT y sólo podemos escribir en los registros que antes llamábamos: sólo de escritura. La siguiente rutina es un ejemplo de cómo hacer dicha operación.

```
IN   A,(&H99)
LD   A,datos a escribir
OUT  (&H99),A
LD   A,número del registro + 128
OUT  (&H99),A
```

La instrucción IN del principio del programa es muy necesaria, porque sincroniza el VDP; más adelante lo veremos con más profundidad. La primera instrucción OUT escribe en el VDP el byte que queramos escribir en el registro.

La siguiente instrucción OUT manda al VDP un número de registro modificado. En determinadas circunstancias, por ejemplo en el caso de una rutina SERVICE INTERRUPTED, la UCP puede ser interrumpida entre las dos instrucciones OUT, lo cual acarrea la subsecuente confusión; ten en cuenta que los componentes del VDP no son muy "listos" que digamos, por lo que debes usar la instrucción IN para borrar cualquier información anterior.

Como ves, es una buena idea ejecutar dicha instrucción antes de pasar a escribir datos por vez primera en el programa del VDP; normalmente yo ejecuto la instrucción IN antes que cualquier acceso al VDP para asegurarme de que el VDP esté preparado para aceptar la siguiente instrucción.

El número de registro que se mandó al VDP tiene 128 unidades más que el número inicial; por ejemplo: para escribir un valor en el registro 4, mandaremos 132 al VDP como número de registro. Esto se hace para prevenir que el número de registro no se confunda con una dirección de la VRAM, ya que éstas están escritas en el VDP de forma parecida.

La siguiente rutina escribirá el valor 18 en el registro 7 del VDP. Si quieres ejecutar el programa, hazlo en el modo de pantalla 0.

```
IN   A,(&H99)
LD   A,18
OUT  (&H99),A
LD   A,135
OUT  (&H99),A
RET
```

Resumiendo: para escribir en un registro VDP se requieren dos instrucciones OUT: la primera, para mandar el byte con los datos, y la segunda, para mandar el número del registro.

Leer del registro de estado del VDP, único registro que podemos leer, es muy fácil. Simplemente tienes que ejecutar una instrucción IN desde la posición &H99 de entrada/salida. El siguiente programa

```
IN    A,(&H99)
LD    (58000),A
RET
```

transferirá el contenido en ese momento del registro de estado VDP a la posición 58000.

Cómo acceder a la VRAM

Acceder a la VRAM es un proceso más laborioso que acceder a los registros, pero, aun así, no es difícil. Aquí también es conveniente leer el registro de estado antes de empezar a escribir en los registros VDP. La dirección de la VRAM se manda al VDP usando dos instrucciones OUT con la dirección &H99. Si quieres volver a leer los datos, se ejecutará una instrucción IN desde la posición &H98 de entrada/salida en vez de una instrucción OUT.

Cómo escribir en la VRAM

```
IN    A,(&H99)
LD    A,VRAM dirección del byte bajo
OUT   (&H99),A
LD    A,VRAM dirección del byte alto + 64
OUT   (&H99),A
LD    A,byte de datos
OUT   (&H98),A
```

Este es el método general para escribir datos en una posición de la VRAM. El efecto que produce escribir datos en una posición de la VRAM depende del modo de pantalla que se use. Fíjate cómo el byte alto de la dirección de la VRAM se modifica antes de que se mande al VDP. En el ejemplo siguiente, esa rutina escribirá la letra "A" en la parte superior izquierda de la pantalla en modo 0.

```
IN    A,(&H99)
XOR   A
OUT   (&H99),A
LD    A,64
OUT   (&H99),A
LD    A,65
OUT   (&H98),A
RET
```

En modo 0, la posición VRAM 0, que es la posición en la que estamos escribiendo en este ejemplo, corresponde a la esquina superior izquierda de la pantalla.

No es muy corriente que sólo escribas un byte en la VRAM; por tanto, sería

bastante útil no tener que asignar la dirección de la VRAM cada vez que quieras mandar un byte a la VRAM siempre y cuando los bytes que vayas a mandar vayan a estar uno detrás de otro en la VRAM; para evitar escribir todas las direcciones, tenemos lo que se llama AUTO INCREMENTACION: una vez que se ha mandado al VDP una dirección, cada dato escrito en la posición &H98, teniendo en cuenta que no se han realizado otros accesos al VDP en ese momento, escribirá el byte de dato en la posición de la VRAM direccionada y después incrementará el puntero interno VRAM del VDP; de esta manera se asigna al VDP la siguiente dirección de la VRAM para la siguiente operación de escritura; así,

```
IN   A,(&H99)
XOR  A
OUT  (&H99),A
LD   A,64
OUT  (&H99),A
LD   A,65
OUT  (&H98),A
OUT  (&H98),A
OUT  (&H98),A
RET
```

escribirá el valor 65 en las posiciones 0, 1 y 2 de la VRAM. Ejecuta este programa en modo 0 para comprobar que funciona, aunque ya lo hemos visto funcionar en el ejemplo 16. Pasemos ahora a ver otro ejemplo de esta AUTO INCREMENTACION, de la que ya te hablé, usando la instrucción OTIR.

EJEMPLO 25

El ordenador MSX tiene dos modos de texto: el modo de pantalla 0 y el 1. Sería muy útil que pudiéramos escribir mensajes en la pantalla como lo hacemos con la instrucción PRINT del BASIC.

Como verás, el ejemplo siguiente es una versión de las variables alfanuméricas del PRINT. Esta rutina sólo se puede usar en modo de pantalla 0, por lo que asegúrate de que el ordenador está en ese modo cuando vayas a ejecutar dicha rutina; este programa tiene una pequeña limitación: siempre escribe la cadena en la misma posición de la pantalla y sólo maneja cadenas de hasta 255 caracteres; esto último lo verás más adelante. Antes de ejecutar esta rutina, no te olvides de que esté en modo 0 tu ordenador.

El argumento de la instrucción USR deberá ser una cadena alfanumérica o una variable. Es decir:

```
1$ = USR("cadena test")
```

escribirá "cadena test" en la pantalla. El programa es como sigue:

```
LD   HL,&HF7F8
LD   C,(HL)
INC  HL
LD   B,(HL)
LD   A,(BC)
LD   (58000),A
INC  BC
LD   A,(BC)
LD   L,A
INC  BC
LD   A,(BC)
LD   H,A
LD   A,(58000)
LD   B,A
INC  A,(&H99)
LD   A,42
OUT  (&H99),A
LD   A,64
OUT  (&H99),A
LD   C,&H98
OTIR
RET
```

Cómo leer de la VRAM

Es posible leer bytes de la VRAM; así, por ejemplo, puedes ver desde el código máquina qué carácter ocupa una determinada posición de la pantalla. Las instrucciones en BASIC para leer datos de la VRAM son las siguientes:

```
IN   A,(&H99)
LD   A,dirección del byte bajo
OUT  (&H99),A
LD   A,dirección del byte alto
OUT  (&H99),A
IN   A,(&H98)
```

Fíjate que no hay modificación de la dirección VRAM que se mandó al VDP, simplemente se manda la dirección como bytes alto y bajo. La siguiente rutina, cuando funciona con la pantalla en modo 0, examina los primeros 960 bytes de la VRAM y sustituye el carácter " " (espacio en blanco) por el carácter ".". Estos bytes contienen la información relacionada con la imagen que sale en pantalla, como seguidamente vamos a ver.

EJEMPLO 26

El código para esta rutina es el siguiente:

```
         LD   BC,0000
         LD   HL,960
BUCLE    IN   A,(&H99)
         LD   A,C
         OUT  (&H99),A
         LD   A,B
         OUT  (&H99),A
```

```
            IN   A,(&H98)
            CP   &H20
            JR   Z,CAMBIO
SINCAMBIO   DEC  HL
            LD   A,H
            OR   L
            RET  Z
            INC  BC
            JR   BUCLE
CAMBIO      IN   A,(&H99)
            LD   A,C
            OUT  (&H99),A
            LD   A,B
            ADD  A,64
            OUT  (&H99),A
            LD   A,&H2E
            OUT  (&H98),A
            JR   SINCAMBIO
```

Ejecuta el programa usando 1 = USR(0), después de introducir el modo de pantalla 0. Date cuenta de que, incluso sin la AUTO INCREMENTACION, este programa es muy rápido. Recuerda que debes sumar 64 al byte alto de la dirección de la VRAM después de escribir en ella, no cuando estés leyendo en ella.

Fíjate en el uso del par de registros BC como contador de la VRAM; es muy fácil enviar una dirección al VDP que esté en dicho par, ya que el byte bajo de la dirección de la VRAM está almacenado en el byte bajo del par de registros.

Con esto se termina la parte, digamos, teórica para escribir en los registros del procesador de video (VDP) y de la VRAM. Veamos ahora cómo utilizar los registros del VDP para controlar la pantalla del MSX.

Las funciones de los registros VDP

En esta sección vamos a usar algunos términos que pueden no ser familiares para ti; me refiero a algunas partes de la VRAM que se explicarán más adelante en este mismo capítulo cuando veamos los modos de pantalla del MSX.

El registro 0

Por ahora sólo los diseñadores del VDP usan los bits 0 y 1 del VDP; de éstos, el bit 1 es muy útil ya que se encarga de la selección del modo de pantalla. También se le llama M3; un poco más adelante veremos su utilización.

El registro 1

Es el registro de control principal del VDP. Para el principiante puede tener algo de caja de Pandora. Algunos bits de este registro son muy útiles, mientras que otros es mejor dejarlos. Examinemos por orden cada uno de ellos.

El bit 7

Es muy aconsejable que lo dejes normalmente a 1; este bit informa al VDP sobre qué *chips* de memoria se han usado para formar la VRAM. Si lo cambias, puede ocurrir que la pantalla sea ilegible.

El bit 6

Normalmente, asígnale un 1; si le asignas un 0 se desactiva la pantalla.

El bit 5

Este bit se denomina DESACTIVADOR DE INTERRUPCIONES DEL VDP, y su estado controla si se genera o no la interrupción del VDP, pero, por si no lo recuerdas bien, informa a la UCP que el teclado necesita leerse, que la variable TIME tiene que actualizarse y algunas otras tareas rutinarias del ordenador que necesitan realizarse.

Si asignas 0 a este bit, la interrupción se desconectará, previniendo así que la UCP realice los trabajos anteriores y, por tanto, desconectando el teclado del BASIC. Podemos seguir leyendo el teclado con acceso directo al PPI.

Los bits 4 y 3

Se denominan también M2 y M1 respectivamente y junto con el M3 controlan el modo de pantalla elegido. La tabla siguiente muestra cómo estos bits controlan los modos.

M1	*M2*	*M3*	*Modo de pantalla*
0	0	0	1
0	0	1	2
0	1	0	3
1	0	0	0

Sin embargo, no creas que puedes cambiar el modo desde tus programas en código máquina solamente cambiando los estados de estos bits; también a los otros registros deben serles asignados los valores apropiados y además debes realizar aún otras cosas. Particularmente, lo que yo hago es asignar el modo de pantalla desde el BASIC antes de introducir un programa en código máquina.

El bit 1

Este bit selecciona el tamaño del SPRITE usado; si se le asigna un 0, entonces se elige el *sprite,* cuyo tamaño es el de una trama de 8 * 8 y si se le asigna un 1 se elige el más grande, es decir, el de una matriz de 16 * 16.

El bit 0

Este bit controla la extensión del *sprite*. Si le asignas a este bit un 0, dará *sprites* sin ampliar y si le asignas un 1, dará *sprites* ampliados.
Sobre los *sprites* hay más información posteriormente.

Valores por defecto para el registro 1

Los valores por defecto en este registro y en los diferentes modos de pantalla son como sigue:

	Decimal	*Binario*
PANTALLA 0	240	11110000
PANTALLA 1	224	11100000
PANTALLA 2	224	11100000
PANTALLA 3	232	11101000

El registro 2

Este registro tiene un valor de 0 a 15, su contenido se multiplica por &H400, da la dirección de comienzo en la VRAM, lo que se llama la TABLA DE NOMBRES, para un determinado modo de pantalla.

Valores por defecto para el registro 2

	Decimal	*Binario*
PANTALLA 0	0	0000
PANTALLA 1	6	0110
PANTALLA 2	6	0110
PANTALLA 3	2	0010

El registro 3

Este registro tiene un valor entre 0 y 255, que, al multiplicarse por &H40, nos da la dirección de comienzo de la TABLA CON LOS COLORES para un determi-

nado modo. Como no hay ninguna elección de color en modo 0, el contenido de este registro es inútil en este modo.

Valores por defecto para el registro 3

Modo de pantalla	*Valor*
0	0
1	128
2	255
3	0

El registro 4

Define la dirección de comienzo en la RAM de la TABLA CON LOS MODELOS para un tipo dado de pantalla. Dicha dirección viene dada al multiplicar el valor del registro por &H800.

Valores por defecto para el registro 4

Modo de pantalla	*Valor*
0	1
1	0
2	3
3	0

El registro 5

El contenido de este registro multiplicado por &H80 da la dirección de la VRAM de la tabla de atributos de los *sprites*.

Valores por defecto para el registro 5

Modo de pantalla	*Valor*
0	0
1	54
2	54
3	54

El registro 6

El contenido de este registro multiplicado por &H800 nos da la dirección de la VRAM de la TABLA DE LOS MODELOS de *sprites*.

Valores por defecto del registro 6

Modo de pantalla	*Valor*
0	0
1	7
2	7
3	7

El registro 7

Es el único registro útil en el tipo de pantalla 0 y controla el color del texto y del fondo de la pantalla. Los cuatro bits altos del registro tienen el código del color de presentación (del texto), y los cuatro bits bajos tienen el código del color para el fondo en modo 0.

Los cuatro bits bajos también tienen el código del color para los márgenes en otros modos de pantalla. Los códigos son los números que se usan en las instrucciones COLOR del BASIC.

Con esto termino el sumario de los registros que he denominado ''sólo para escribir''. Pasemos ahora al registro de estado del VDP.

El registro de estado

El bit 7

Es el *flag* de interrupción, y se encarga de generar la interrupción VDP. Siempre que al bit de activación de la interrupción del registro 1 del VDP se le asigne un 1 se enviará una señal de interrupción. Normalmente, este bit lo borra la rutina de TRATAMIENTO DE INTERRUPCIONES leyendo del registro de estado. Si no se borra el bit de esa forma, las interrupciones siguientes no serán respondidas por la UCP y tendrás problemas.

Sólo tienes que ocuparte de asignar un 0 a este bit en respuesta a las interrupciones, si te deshaces de la rutina normal de TRATAMIENTO DE INTERRUPCIONES, lo cual no es una buena idea para principiantes.

El bit 5

Este es el *flag* de COINCIDENCIA del *sprite,* y se le asigna un 1 siempre que dos *sprites* coinciden con la pantalla. Incluso si hay solo dos componentes del *sprite* coincidiendo, este *flag* se activa. Puede ser examinado por el programador. Recuerda que este bit vuelve a 0 después de una lectura al registro de estado.

El bit 6

Normalmente, el sistema de video del MSX permite sólo cuatro *sprites* en cada línea horizontal de la pantalla. Si resulta que hay cinco *sprites* intentando ocupar la misma línea, este bit se pondrá a 1. Los cuatro bits bajos de este registro tienen el número del plano del *sprite* que ha causado el problema.

Si todavía no estás familiarizado con las instrucciones en BASIC que he usado para acceder al VDP te aconsejo que consultes el manual de tu MSX *(Descubre tu MSX)* o un libro de este tipo, ya que no estudiaremos dichas instrucciones en este capítulo con más detalle.

Antes de explicar el método de posicionamiento en la VRAM con los distintos modos de pantalla, veamos un ejemplo de cómo escribir en los registros del VDP. Puede que pienses que usando la instrucción VDP podrás leer datos de los registros VDP ''sólo para escribir''; pero esto es imposible; entonces, ¿cómo hace esto el BASIC?

La respuesta es muy simple: una copia del contenido de los registros del VDP se guarda en la zona de trabajo del sistema, en las posiciones de la 62431 a la 62438. La posición 62431 tiene una copia del contenido del registro 0 y la posición 62438 del contenido del registro 7. Estas copias de la RAM sólo se actualizan con el sistema operativo cuando cambiemos de modo de pantalla o con la instrucción VDP(n) = variable.

Si escribes datos en los registros del VDP sin usar la instrucción VDP, no se actualizarán dichas copias de la RAM y puede ocasionarte problemas a la hora de volver del BASIC si las copias de la RAM no concuerdan con el contenido del registro; por tanto, sería conveniente actualizar la copia de la RAM de un registro cuando modifiques éste.

Pasemos ahora a utilizar el registro 4 del VDP. En modo 0 tiene la dirección de comienzo de la tabla con los modelos, que informa al VDP sobre la imagen que va a salir en la pantalla por cada carácter. Esto lo veremos más adelante.

Si modificas este registro, el procesador de video cogerá la información, que en ese caso necesita de un área diferente de la VRAM, y, por consiguiente, los caracteres que aparezcan en la pantalla serán diferentes. Sería interesante intercambiar dos valores diferentes en el registro 4 y ver los resultados.

EJEMPLO 27

Esta rutina sólo funciona en modo 0, así que ejecuta una instrucción SCREEN 0 antes de lanzarla. Si pulsas la barra espaciadora cuando el programa se esté ejecu-

tando, intercambiará las dos direcciones de la tabla de los patrones, y cuando termines de pulsar la barra, la pantalla será legible para la nueva tabla de patrones.

Cuando quieras volver al BASIC, pulsa la tecla DEL. El siguiente programa NO usa, a propósito, rutinas y sólo es una especie de ejercicio para que tú reescribas el código usando subrutinas.

```
        IN   A,(&H99)
BUCLE   LD   A,&HF8
        OUT  (&HAA),A
        IN   A,(&HA9)
        BIT  3,A              ;comprueba si se pulsa la tecla DELETE
        RET  Z                ;si esta pulsada vuelve al BASIC
        CP   255              ;si esta pulsada alguna tecla
        JR   Z,BUCLE          ;si no, espera a que asi sea
        LD   A,2
        OUT  (&H99),A         ; envia el byte con el dato
        LD   A,&H84
        OUT  (&H99),A         ; envia el numero de registro
BUCLE1  LD   A,&HF8
        OUT  (&HAA),A
        IN   A,(&HA9)
        BIT  3,A
        RET  Z
        CP   255
        JR   Z,BUCLE1
        LD   A,1
        OUT  (&H99),A         ;envia un nuevo valor al registro
        LD   A,&H84
        OUT  (&H99),A         ;envia el numero de registro
        JR   BUCLE
```

Además de enseñarte a escribir en los registros del VDP, esta rutina te muestra cómo controlar el teclado con programas en código máquina.

Pasemos ahora a estudiar cómo está dispuesta la VRAM en los diferentes modos de pantalla: lo primero que deberemos hacer es definir algunos de los términos y expresiones que te encontrarás en este capítulo.

Tabla de nombres

Esta es una zona de la VRAM que indica al VDP qué imagen va a aparecer en la pantalla en cualquier posición. Una entrada de la tabla de nombres es un número entre 0 y 255 y se usa como índice de las tablas con los patrones y colores para saber cuál es la imagen que corresponde a esa entrada de la tabla de nombre. La forma de utilizarla depende del modo de pantalla que estés usando.

Por ejemplo, en modo 0, cada entrada es el código ASCII del carácter que quieres que aparezca en el punto de la pantalla correspondiente a esa entrada de la tabla de nombres. Usando la instrucción BASE( ) del BASIC hallamos la dirección de comienzo de la tabla de nombre en la VRAM y en los distintos modos de pantalla.

BASE (0) DIRECCION DE COMIENZO EN MODO 0
BASE (5) DIRECCION DE COMIENZO EN MODO 1
BASE (10) DIRECCION DE COMIENZO EN MODO 2
BASE (15) DIRECCION DE COMIENZO EN MODO 3

La instrucción

PRINT BASE(n)

devuelve la dirección correspondiente, y así podrás usarla en tus programas en código máquina.

Tabla de patrones

En todos los modos esta tabla indica la imagen que va a mostrarse en la pantalla para cada entrada concreta en la tabla de los nombres.

Este punto lo veremos más adelante con más detalle cuando explique cada modo de pantalla.

BASE (2) DIRECCION DE COMIENZO EN MODO 0
BASE (7) DIRECCION DE COMIENZO EN MODO 1
BASE (12) DIRECCION DE COMIENZO EN MODO 2
BASE (17) DIRECCION DE COMIENZO EN MODO 3

Tabla de los colores

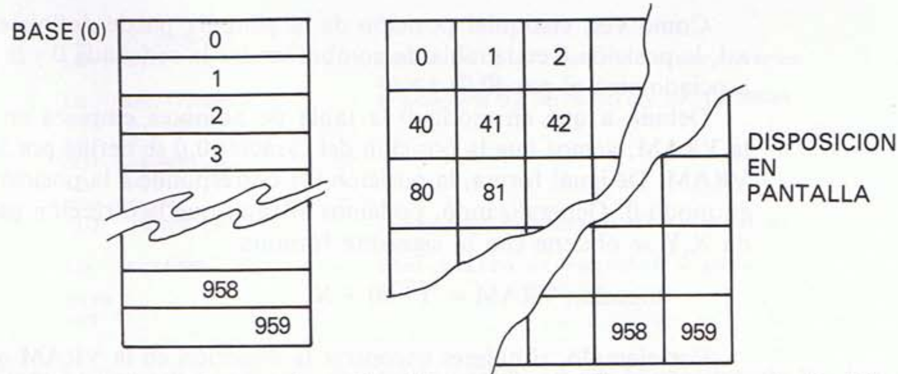
Esta tabla indica al VDP qué colores se van a usar para cada entrada de la tabla de nombres. No en todos los modos hay tablas de colores:

BASE (6) DIRECCION DE COMIENZO EN MODO 1
BASE (11) DIRECCION DE COMIENZO EN MODO 2

Las dos tablas relacionadas con los *sprites,* es decir, la tabla de atributos de los *sprites* y la tabla de patrones de los *sprites,* o tabla generadora, se estudiarán en el capítulo dedicado a los *sprites.*

Modo 0

El modo 0 de pantalla se selecciona ejecutando la instrucción SCREEN 0 del BASIC y es un modo de texto de 40 * 24. Se necesitan sólo dos tablas de la VRAM para ejecutar este modo; me estoy refiriendo a la tabla de nombres y a la de patrones. La tabla de nombres en modo 0 ocupa 960 bytes y aparece en la pantalla de la manera siguiente:



En este modo, cada entrada en la tabla con los nombres corresponde al código ASCII del carácter que quieres que aparezca en la pantalla. Por ejemplo: si asignas a la posición 0 de la tabla de nombre el valor 65, en la parte superior izquierda de la pantalla saldrá una "A".

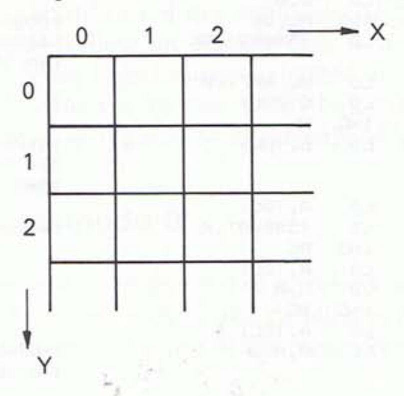
Hacer esto es muy fácil, ¿verdad?, además ya hemos visto ejemplos de esta operación en los ejemplos 4 y 16 anteriores. Si seguimos asignando valores en las restantes posiciones de la tabla de nombres, aparecerán más caracteres en la pantalla; por ejemplo, si asignamos el valor 66 a la posición 1 de la tabla de nombre, aparecería una "B" al lado de la "A" anterior.

Para ver qué caracteres salen en la pantalla en modo 0, ejecuta el programa en BASIC.

```
10 FOR I=0 TO 255
20 PRINT CHR$(I);
30 NEXT I
```

Hay muchos caracteres del programa anterior que nos ofrecen la posibilidad de crear gráficos de baja resolución en el modo de pantalla 0, además, se pueden escribir cadenas de los caracteres anteriores en la pantalla. Ya vimos un pequeño ejemplo de escribir cadenas en pantalla con un programa en el ejemplo 25; ahora vamos a ver otro ejemplo que nos ofrece más posibilidades a la hora de presentar texto en la pantalla.

Empieza numerando las posiciones de la pantalla, como sigue:



Como ves, cualquier posición de la pantalla puede definirse en coordenadas; así, la posición 0 en la tabla de nombre tendrá la ordenada 0 y la abscisa 0 y su par asociado será el par (0,0).

Debido a que en modo 0 la tabla de nombres empieza en la posición 0 de la VRAM, vemos que la posición del carácter 0,0 se define por la posición 0 de la VRAM. De igual forma, la posición 0,1 corresponde a la posición 40 de la VRAM en modo 0. Generalizando, podemos afirmar que la dirección para una coordenada X,Y se obtiene con la siguiente fórmula.

dirección VRAM = $Y * 40 + X$

Por ejemplo, si quieres encontrar la dirección en la VRAM que corresponde a la posición de pantalla 2,1, simplemente tienes que sustituir dichos valores en la fórmula anterior como sigue:

$1 * 40 + 2 = 42$

Fíjate que las posiciones en pantalla que estoy utilizando aquí son absolutas; no se modificarán por la instrucción BASIC WIDTH, con lo que podremos escribir una rutina en código máquina para posicionar una cadena en una determinada posición en la pantalla y en modo 0.

EJEMPLO 28

Esta rutina tiene que introducirse a partir de la dirección 59000. Recuerda que el modo 0 de la tabla de nombres empieza en la dirección 0 en la VRAM.

```
      LD   B,39                ;para multiplicar por 40
      LD   A,(58001)           ;coordenada Y
      LD   E,A
      LD   D,0
      LD   L,A
      LD   H,0
BUCLE ADD  HL,DE
      DJNZ BUCLE
      LD   A,(58002)
      LD   E,A
      LD   D,0
      ADD  HL,DE               ;coge el valor X sumado
      LD   (58002),HL          ;almacena el  resultado  en 58002 y
                               ;en 58003
      LD   HL,&HF7F8
      LD   C,(HL)
      INC  HL
      LD   B,(HL)              ;direccion de la descripcion de
                               ;la cadena
                               ;dentro del par de registros BC
      LD   A,(BC)
      LD   (58000),A           ;almacena la longitud de la cadena
      INC  BC
      LD   A,(BC)
      LD   L,A
      INC  BC
      LD   A,(BC)
      LD   H,A                 ;mete la direccion de la cadena
                               ;en HL para OTIR
```

```
LD    A,(58000)
LD    B,A                    ;mete en B el numero de caracteres
                             ;para OTIR
LD    DE,(58002)             ;recupera la direccion de la VRAM
IN    A,(&H99)
LD    A,E
OUT   (&H99),A
LD    A,D
ADD   A,64
OUT   (&H99),A               ;envia la direccion de la VRAM al
                             ;VDP
LD    C,&H98                 ;inicializa el registro C para
                             ;OTIR
OTIR
RET
```

Para poder usar esta rutina, la posición 58001 tendrá la coordenada Y y la posición 58002, la coordenada X; para ejecutarla, ten en cuenta que debes haber usado la instrucción DEFUSR0 para asignar la dirección de comienzo de la rutina.

POKE 58002,10: POKE 58001,10: 1$ = USR("CADENA DE PRUEBA")

escribirá la cadena en la posición 10,10 de la pantalla. Recuerda que las coordenadas anteriores tienen el VDP como referencia y no la del BASIC del MSX.

Hay dos cosas de la rutina anterior que me parece importante señalar:

a) Las cadenas no deben ocupar más de 255 caracteres, debido a que usamos la instrucción OTIR.

b) Es posible, asignando coordenadas "falsas", escribir una cadena en una zona de la VRAM que NO sea la tabla de nombres. Sería conveniente, pues, que te ideases una rutina que compruebe las coordenadas que vayan a ir al programa anterior, teniendo en cuenta que a veces las cadenas que vas a utilizar son muy largas y las direcciones de la VRAM que vas a usar pueden estar casi al final de la tabla de nombre, por lo que es posible escribir en zonas de la VRAM no deseadas, aun cuando tus coordenadas sean correctas; por eso, y como te he dicho antes, debes idearte algún tipo de control para las coordenadas que introduzcas en la rutina.

¿Qué pasa con los colores en modo 0? Ya hemos visto anteriormente que no hay tablas de color en este modo; el color se define como el contenido del registro 7 del VDP; repasa el capítulo en el que expliqué este registro, para más información; aquí sólo te voy a explicar un ejemplo que va a consistir en poner texto negro en un fondo verde. Para lograr esto, asignamos un 1 (que es el código para el negro) a los cuatro bits altos y a los cuatro bits bajos les asigno el valor 2, que es el código para el verde, y al registro se le asignará el valor 18.

Tabla de modelos en modo 0

Esta tabla ocupa 2048 bytes y en modo 0 se posiciona en la dirección 2048 de la VRAM. Se divide en 256 bloques y cada uno de ellos tiene 8 bytes. Cada bloque forma la definición del carácter para uno de los caracteres en ASCII que aparecen en la pantalla.

Por ejemplo, los bytes que forman la definición del carácter para la letra "A" son: 32, 80, 136, 136, 248, 136, 136, 0. Estos números escritos en decimal no significan nada, pero si los pasamos a binario en el mismo orden en que están en la tabla con los modelos, queda algo como lo siguiente:

00100000
01010000
10001000
11111000
10001000
10001000
00000000

Si te fijas, te darás cuenta de que los unos en binario forman una "A" y los ceros son, en este caso, el fondo negro para esa letra.

Todos los demás caracteres se definen de la misma forma. Los primeros 8 bytes de la tabla modelo tienen la definición para CHR$(0); los segundos, para la CHR$(1), etc. De esta forma, cuando en una posición de la tabla de nombre tiene el valor el carácter definido en los primeros 8 bytes de la tabla modelo, se mostrará en esa posición de la tabla de nombres.

De esta forma los 8 bytes de la VRAM que empiezan por

(65 * 8) + 2048

en modo 0 tienen la definición de la letra "A", cuyo código ASCII es 65. En modo 0, la ROM del MSX llena todas las posiciones de la tabla de nombres con el valor 32; así se llena toda la pantalla de espacios vacíos.

Cuando ejecutes la instrucción

SCREEN 0

todas las definiciones de los caracteres en este modo se copian de la MSX ROM y se usan para definir los caracteres que vemos normalmente.

Las definiciones de la ROM se almacenan desde la posición 7103 en adelante. Desde nuestro punto de vista esto es muy conveniente, ya que las definiciones que usa el VDP se almacenan en la VRAM, y como podemos modificar zonas de la VRAM, también podremos modificar los caracteres que aparecen en la pantalla modificando sus definiciones. Por ejemplo: vamos a cambiar la definición del carácter espacio; lo primero que deberemos saber es dónde empieza su definición en la tabla de patrones; como ya sabes, el comienzo de cualquier definición de caracteres en la RAM y en modo 0 viene dada por

comienzo = (n * 8) + 2048

donde *n* es el código ASCII del carácter buscado y 2048 es la dirección de comienzo de la tabla con los patrones en la VRAM. Puedes usar también esta fórmula

con otros modos simplemente cambiando la dirección de comienzo de la tabla con los patrones, con el valor correcto para ese modo.

Ahora ya es muy fácil saber cuál de los siguientes bytes necesitan modificarse para cambiar un carácter. En el ejemplo siguiente vamos a modificar el último byte de la definición del carácter que corresponde a la parte más baja del carácter en la pantalla.

EJEMPLO 29

Esta rutina es muy fácil de seguir: el número 2311 es la dirección en la VRAM del último byte que define el espacio en blanco. Si quieres, puedes ejecutar este programa para ver lo que ocurre:

```
LD   DE,2311
IN   A,(&H99)
LD   A,E
OUT  (&H99),A
LD   A,D
ADD  A,64
OUT  (&H99),A
LD   A,255                      ;nuevo valor del byte
OUT  (&H98),A
RET
```

También puedes escribir este mismo programa usando las instrucciones OTIR para modificar los distintos bytes de definición de un carácter. No te olvides de apuntar el registro HL a la nueva definición en la RAM y carga el registro B con el número necesario de bytes.

Cualquier carácter, como ves, puede ser redefinido; es mucho más útil redefinir aquellos caracteres cuyos códigos ASCII tienen valores más bien altos. Yo, normalmente, escojo códigos 240 en adelante, con lo que prevengo posibles caracteres extraños en los listados. Por ejemplo, este uso de la definición de caracteres es muy útil para lenguajes extranjeros, etc.

Pero, como con casi todas las ideas muy simples, hay una pequeña falacia: cada vez que cambias el modo y después vuelves al modo 0 incluso ejecutando la instrucción

SCREEN 0

mientras esté en modo 0, el conjunto de caracteres de la ROM se escribirá en la VRAM, y, por tanto, todo tu trabajo se borrará. Por esto, si quieres usar esa técnica, tendrás que hacerte una pequeña rutina que copie de la RAM tu conjunto de caracteres modificados, lo cual no es muy difícil, siempre y cuando recuerdes que la instrucción OTIR sólo transfiere 256 bytes a la vez. Si quieres mandar, por ejemplo, un bloque de datos de 2048 BYTES para redefinir todo el conjunto de caracteres, tendrías que usar varias instrucciones OTIR.

Ten en cuenta que la disposición de la VRAM en modo 0 no utiliza la zona por encima de la tabla de patrones. Como en modo 0 no se utilizan *sprites,* está

desaprovechada; normalmente, no se usa el resto de VRAM. Sin embargo, puedes asignar conjunto de caracteres alternativos en esta área en modo de nuevas tablas de patrones y sólo tendrías que modificar el registro del VDP necesario para ganar acceso, como vimos en el ejemplo 27. También podrías definir tablas de nombres alternativas.

Modo 1

Este modo es también sólo para texto, pero permite usar *sprites*. Hay tres tablas en la VRAM: la tabla de nombre, la de los patrones y la de color; además, hay dos tablas de *sprites,* que veremos más adelante.

Tabla de nombres

La tabla de nombres en modo 1, que realiza la misma función que en modo 0, se sitúa en la dirección 6144 de la VRAM y ocupa 768 bytes; esto explica por qué no podemos introducir tantos caracteres en una línea del texto en modo 1 como en una línea en modo 0.

Aparte de ocupar bytes diferentes, se dispone de manera parecida a como lo hacía en modo 0, es decir, cada entrada de la tabla se corresponde con una posición de la pantalla; o sea, que, para asignar un carácter en la pantalla, sólo tienes que cargar la posición de la tabla de nombre apropiada con el código ASCII del carácter. De esta manera, todos los ejemplos de programas que hemos visto en modo de pantalla 0 también los puedes utilizar en modo 1, por lo que te aconsejo seas tú el que los uses en vez de darte yo todo el trabajo hecho.

Para pasarlos a modo 1, recuerda los siguientes consejos:

1. La nueva dirección de comienzo de la tabla de nombre es la 6144 y no la 0.
2. La nueva tabla de nombres ocupa 768 bytes y no 960 y se dispone en una matriz 32 * 24.

En el ejemplo 28, por tanto, ejecutamos una multiplicación por 32 en vez de por 40, es decir, que sustituiremos la primera instrucción del programa por LD B,31.

Fíjate también que el REGISTRO 7 no tiene nada que ver con el modo 1 del texto o con el color del fondo. Los bits del registro 7 del VDP que normalmente controlan el color del fondo en modo 0 definen el color de los márgenes en otros modos.

Tabla de patrones

Esta tabla de patrones es como las otras tablas que hemos visto del mismo nombre, excepto por la posición que ocupan en la VRAM. Esta tabla en modo 1 ocupa 2048 bytes y empieza en la posición 0 de la VRAM.

Se dispone igual que la tabla de patrones en modo 0, es decir, en 256 bloques de 8 bytes y cada bloque define un carácter. Si quieres redefinir los caracteres en este modo, haz lo mismo que hicimos en modo 0, pero ahora ya no sumes 2048 para obtener la dirección del byte que quieres modificar en la VRAM. La posición VRAM (8 * 32) = 256 es el primer byte de la definición para el carácter de espacio. El conjunto de caracteres también se copia de la ROM siempre que se entre en este modo.

Tabla de colores

Esta tabla empieza en la dirección 8192 de la VRAM y ocupa la ridícula cifra de 32 bytes. En las explicaciones del MSX de tu manual puede que hayas leído que el modo 1 es, como el modo 0, un modo de dos colores (bicolor). Esto no es totalmente cierto, ya que es posible hasta obtener 16 colores a la vez.

Si nos fijamos en los pocos bytes que esta tabla ocupa, parece lógico pensar que no sea posible definir una combinación diferente de color por cada elemento de la tabla de nombres; lo que ocurre es que cada byte de la tabla de colores tiene el color para las definiciones de un conjunto de 8 caracteres en la tabla de modelos.

Por ejemplo, el primer byte de la tabla de colores tiene el color de los caracteres en código ASCII del 0 al 7; el segundo byte de los siguientes 8 caracteres, etc.; el último byte tendrá los colores para los caracteres del 248 al 255. Si modificas el contenido de esta tabla, obtendrás 16 colores a la vez en la pantalla, pero las combinaciones de los colores se asignan en determinados caracteres.

La información de los colores en la tabla se dispone igual a como vimos en la del registro 7 del VDP. Intenta escribir la tabla con los colores para ver los efectos generados.

Acuérdate de leer el registro de estado del VDP antes de realizar cualquier operación de escritura en el VDP.

Con esto doy por terminados los dos modos de sólo texto. Aunque puedes crear gráficos muy simples en los dos (redefiniendo el conjunto de caracteres), estos modos no fueron creados para desempeñar dicha función.

Pasemos ahora a ver los dos modos de gráficos del MSX. Debido a la disponibilidad de *sprites* en los ordenadores MSX, utilizamos el BASIC para que haga el trabajo más pesado (movimiento de los *sprites* en la pantalla), ya que éstos son muy fáciles de programar en BASIC, no siendo así en código máquina. No me voy a parar mucho en el movimiento; simplemente te indicaré cómo crear algunos gráficos sencillos en la pantalla.

El modo 2

El modo de pantalla 2 nos ofrece gráficos de alta resolución, 16 colores en la pantalla a la vez y además... ¡un buen dolor de cabeza! No es un modo fácil de programar como lo eran los anteriores. La disposición de la VRAM en este modo

tampoco es tan fácil de entender como lo era en modo 0 y en modo 1. Por tanto, mi consejo es que prestes mucha atención a este capítulo, y no creo que tengas problemas con este modo.

Tabla de nombres

La tabla de nombres en modo 2 empieza en la dirección 6144 de la VRAM y ocupa 768 bytes. En la pantalla se dispone de la misma manera que expliqué en modo 1 en la VRAM. Sin embargo, hay una gran diferencia en cómo esta tabla en modo 2 define la imagen que va a aparecer en la pantalla; en los modos 0 y 1 el valor de una entrada particular de la tabla de nombres se define directamente al acceder a la zona apropiada de la tabla de patrones y ésta aparece en la pantalla.

En modo de pantalla 2, la combinación del valor de la posición en la tabla de nombres junto con la posición dentro de la tabla son los que definen la imagen que va a aparecer en la pantalla.

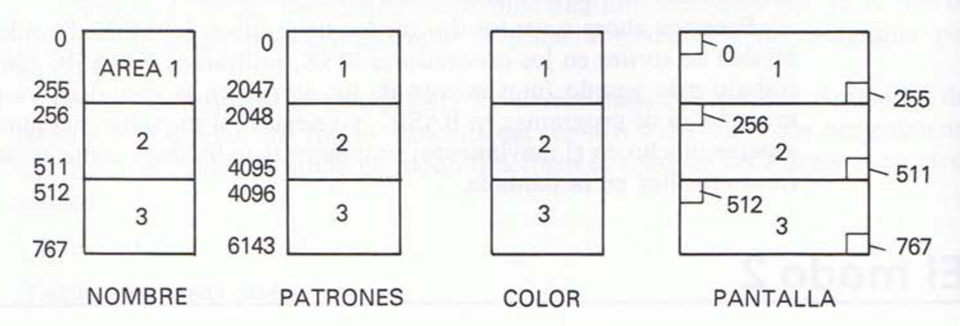
Quizá esto te parezca un poco complicado; no te preocupes, y sigue leyendo las otras dos tablas, que te clarificarán tus posibles dudas.

Tabla de patrones

Esta tabla es grandísima, ya que empieza en la posición 0 de la VRAM y ocupa 6144 bytes. Permite que cada posición de la pantalla contenga una imagen diferente a las restantes de la pantalla.

Tabla de colores

Esta tabla también es muy larga y ocupa 6144 bytes, empieza en la posición 8192 de la VRAM. Seguidamente te facilito la pantalla con respecto a las tablas de memoria:



Aunque esta tabla es mucho más grande que la tabla de color en modo 0, se dispone de manera similar, es decir: en bloques de 8 bytes, donde cada bloque representa una imagen que puede ser representada en la pantalla. Los primeros

2048 bytes de la tabla de patrones definen las imágenes que aparecen para los primeros 256 valores que se puedan asignar a la tabla de nombres, es decir, la zona de la tabla de nombres del diagrama anterior llamada AREA 1. Para el AREA 2 de la pantalla, la imagen que va a salir en la pantalla se define con el AREA 2 de la tabla de nombres, que accede a la correspondiente área de la tabla de patrones.

Algo parecido ocurre para el área 3 de la pantalla, que se corresponde con el área 3 de las tablas de nombres y de patrones.

Veamos ahora un ejemplo sencillo:

La posición 0 de la tabla de nombres tiene el valor 0, que hará que aparezca, en la posición 0 de la pantalla, la imagen definida por los 8 primeros bytes del AREA 1 de la tabla de patrones. Sin embargo, si cargas la posición 256 de la tabla de nombres con ceros, causará que la imagen definida en los primeros 8 bytes del área 2 de la tabla de patrones aparezca en la posición 256; por tanto, la imagen que aparezca en esa posición será aquella definida en los 8 bytes a partir de la posición 2048 de la tabla de patrones.

Tabla de color

Se dispone de forma parecida a la tabla de patrones en este modo. Cada byte de la tabla de colores define la combinación de colores para un solo byte de la tabla de patrones.

Así es como se consigue una resolución de colores en modo 2 con ocho colores verticalmente y dos colores distintos horizontalmente por cada posición de cada carácter en la pantalla.

Los 8 bytes a partir de la posición 0 de la tabla de colores definen los colores que se van a usar con el correspondiente elemento de la tabla de patrones. Los colores se definen igual como dije en modo 0. Vuelve a leer el apartado referente al registro 7 del VDP si tienes alguna duda.

Debido a que el conjunto de caracteres de la ROM no se copia en la tabla de patrones, todas las entradas de la tabla de patrones que queramos usar tendrán que ser definidas por nosotros en código máquina. Cuando usamos instrucciones del BASIC para dibujar o escribir texto en la pantalla, el sistema operativo se encarga de lo que te acabo de explicar.

La única inicialización que se realiza al entrar en este modo es que a cada entrada en la tabla de nombres se le asigna el valor correspondiente a su posición dentro de la zona de la pantalla apropiada; así, el primer elemento del área 1 de la tabla de nombres tendrá el valor 0; la segunda, el 1, etc. Algo parecido ocurre también en las áreas 2 y 3 de la tabla de nombres.

El resultado en la práctica de asignar valores es que, si cargas un valor en la correspondiente área de la tabla de patrones, verás la correspondiente posición en la pantalla con la imagen previamente definida.

EJEMPLO 30

Este es un ejemplo de lo dicho anteriormente. Vamos a llevar una imagen a las posiciones 1.257 y 513 de la pantalla. Para hacer esto, escribiremos en las posicio-

nes apropiadas de la tabla de patrones, lo cual quiere decir que tendremos que escribir en las posiciones pertinentes de la tabla de patrones que corresponda a la segunda entrada de cada área de la tabla de nombres; por tanto, tendremos que escribir en las posiciones 8, 2056 y 4140 de la VRAM para escribir datos en el primer byte de la entrada correspondiente en la tabla de patrones.

```
        IN   A,(&H99)
        LD   DE,08
        CALL SALIDA
        LD   DE,2056
        CALL SALIDA
        LD   DE,4104
        CALL SALIDA
        RET
SALIDA  LD   A,E
        OUT  (&H99),A
        LD   A,D
        ADD  A,64
        OUT  (&H99),A
        LD   A,255                ;valor a escribir en la entrada
                                  ;de la tabla de patrones
        OUT  (&H98),A
        RET
```

Ejecuta este programa en BASIC:

```
10 DEFUSR=DIRECCION DEL PROGRAMA
20 SCREEN 2
30 I=USR(0)
40 GOTO 40:REM PREVIENE REGRESAR AL MODO DE TEXTO
```

En la pantalla aparecerán tres líneas señalando las posiciones en la pantalla que corresponden a las entradas en la tabla de patrones que hayas modificado. Si, por ejemplo, quisiéramos asignarles a todas las entradas de la tabla de nombres el valor 1, acabaríamos con una pantalla llena de líneas. Esto debería ser obvio para ti; recuerda que las definiciones escritas en la tabla de patrones corresponden a las entradas de 1 en la tabla de nombres. El siguiente programa es un ejemplo de esto último que te acabo de explicar.

EJEMPLO 31

Este programa dibuja una línea horizontal en la pantalla:

```
DI                  ;Desactiva interrupciones
IN      A,(&H99)
LD      DE,08       ;Define el area 1
CALL    SALIDA
LD      DE,2056     ;Define el area 2
CALL    SALIDA
LD      DE,4104     ;Define el area 3
CALL    SALIDA
LD      DE,6144     ;Comienzo de la tabla de nombres
LD      HL,768      ;Longitud de la tabla
```

```
BUCLE   LD      A,E
        OUT     (&H99),A
        LD      A,D
        ADD     A,64
        OUT     (&H99),A
        LD      A,1
        OUT     (&H98),A        ;Envia valor a la tabla de nombres
        INC     DE
        DEC     HL
        LD      A,L
        OR      H
        JR      NZ,BUCLE        ;Hemos enviado todos los bytes?
        EI                      ;Si, activa interrupciones y vuelve al BASIC
        RET
SALIDA  LD A,E
        OUT     (&H99),A
        LD      A,D
        ADD     A,64
        OUT     (&H99),A
        LD      A,255
        OUT     (&H98),A
        RET
```

Las interrupciones se quitaron para prevenir posibles intervalos en las líneas; al ensamblar el código se escribirán los bytes de manera que te resulte fácil localizar la posición de la subrutina SALIDA. Si escribes el código en la dirección 59000, la subrutina SALIDA estará en la posición &HE6A7.

¿Cómo puedes dibujar una línea horizontal?

ENTRADA A LA TABLA DE PATRONES ↓

0	8	16
1	9	17
2	10	18
3	11	19
4	12	20
5	13	21
6	14	22
7	15	23

ENTRADA 1 A LA TABLA DE NOMBRES → (fila 6)

↑ ENTRADA A LA TABLA DE NOMBRES

Los bytes del 0 al 255 de la tabla de patrones corresponden a la línea superior de la pantalla. También puedes dibujar una línea horizontal, por ejemplo, en la mitad de la pantalla y que la cruce toda, escribiendo 255 en las posiciones de la tabla de patrones 4, 12, 20, etcétera. Recuerda que la tabla de nombres se inicializará como vimos anteriormente. De ahora en adelante dejaremos igual la tabla de nombres cuando usemos el modo de pantalla 2 y manejaremos la tabla de patrones para cambiar las imágenes de la pantalla.

Más adelante veremos algunas excepciones.

EJEMPLO 32

Esta rutina dibuja una línea horizontal de un lado al otro de la pantalla; además, podrás elegir la posición del eje Y de la pantalla donde quieres dibujar la línea. Ejecuta la rutina en modo 2 después de ejecutar las siguientes instrucciones POKE:

POKE 58000, NUMERO DE LA LINEA VERTICAL (0—23).
POKE 58001, PIXEL DENTRO DE LA LINEA (0—7).

Para nuestro ejemplo, será

POKE 58000,0 : POKE 58001,4

El programa es el siguiente:

```
        DI
        IN   A,(&H99)
        LD   A,(58000)
        LD   H,A
        LD   A,(58001)
        LD   L,A                       ;mete la direccion de
                                       ;comienzo de la linea en el
                                       ;par de registros HL
        LD   DE,08                     ;incrementa
        LD   BC,32                     ;numero de caracteres que
                                       ;seran dibujados
BUCLE   LD   A,L
        OUT  (&H99),A
        LD   A,H
        ADD  A,64
        OUT  (&H99),A
        LD   A,255
        OUT  (&H98),A
        ADD  HL,DE
        DEC  BC
        LD   A,B
        OR   C
        JR   NZ,BUCLE
        EI
        RET
```

Con esta rutina, también podremos dibujar una línea desde el margen izquierdo hasta cualquier posición de esa línea con una resolución horizontal de ocho *pixels*.

El siguiente ejemplo es un caso práctico de lo que te acabo de explicar:

EJEMPLO 33

Se usan tres instrucciones POKE para que funcione esta rutina:

POKE 58000,(INT(Y/8))
POKE 58001,8 * ((Y/8) — INT(Y/8))
POKE 58002,X/8

Y = COORDENADA Y DEL GRAFICO
X = COORDENADA X DEL FINAL DE LA LINEA

```
        DI
        IN   A,(&H99)
        LD   A,(58000)
        LD   H,A
        LD   A,(58001)
        LD   L,A
        LD   DE,8
        LD   A,(58002)              ;obtiene la longitud de
                                    ;la linea
        LD   C,A
        LD   B,0
BUCLE   LD   A,L
        OUT  (&H99),A
        LD   A,H
        ADD  A,64
        OUT  (&H99),A
        LD   A,255
        OUT  (&H98),A
        ADD  HL,DE
        DEC  BC
        LD   A,B
        OR   C
        JR   NZ,BUCLE
        EI
        RET
```

Si ahora quieres escribir un programa que acepte la abscisa X para poder escribir líneas horizontales por toda la pantalla en código máquina y conseguir una resolución mayor en el eje de las X que la resolución de 8 *pixels* que estamos usando, sigue leyendo.

De igual forma a como hemos escrito otras líneas, también podemos escribir líneas verticales. Antes de pasar a reescribir las rutinas DRAW y LINE del MSX, te diré que, normalmente, la mayoría de ordenadores usan el código máquina para crear gráficos móviles, pero en los ordenadores MSX tenemos los *sprites* que nos facilitan ese trabajo; por tanto, particularmente, suelo escribir muy pocos programas en código máquina que dibujen gráficos; normalmente, dibujo los fondos fijos en BASIC y el código máquina lo utilizo para controlar el movimiento de los *sprites*. De todas maneras, este tema lo estudiaremos con mayor profundidad más adelante, aunque en este mismo capítulo.

Los colores

Los colores en modo 2 se modifican cambiando el área de la tabla de colores que corresponda a la imagen de la pantalla que queramos colorear. Este punto ya lo he explicado anteriormente, por lo que concluyo la explicación de la pantalla en modo 2 con un ejemplo que accede a las tablas con los colores, nombres y patrones a la vez.

EJEMPLO 34

La siguiente rutina llena el tercio superior de la pantalla con pequeñas figuras negras en un fondo verde.

```
         DI
         IN   A,(&H99)
         LD   HL,TABLA           ;asigna HL para que señale la
                                 ;definicion del caracter
         LD   B,8                ;numero de bytes
         LD   DE,0               ;direccion de la VRAM de la
                                 ;tabla de modelos
         CALL SALIDA
         LD   C,&H98
         OTIR                    ;envia la definicion del
                                 ;caracter a la tabla de patrones
         LD   DE,8192
         LD   B,8
         CALL SALIDA
         LD   A,&H12             ;escribe los valores pertinen-
                                 ;tes en la zona adecuada de la
                                 ;tabla de color
BUCLE    OUT  (&H98),A
         DJNZ BUCLE
         LD   DE,6144
         LD   B,0                ;carga 256 en B
         CALL SALIDA
         XOR  A
         BUCLE 2 OUT  (&H98),A
         DJNZ BUCLE2
         EI
         RET
SALIDA   LD   A,E
         OUT  (&H99),A
         LD   A,D
         ADD  A,64
         OUT  (&H99),A
         RET
```

Ejecuta este programa en modo 2 y no te olvides de insertar algunas instrucciones en BASIC para evitar que el ordenador vuelva al modo de texto y así puedas ver el resultado de tu trabajo.

Con este programa termino la explicación de los gráficos en modo 2, ahora vamos a ver lo que se puede hacer en el modo de pantalla 3.

Modo 3

El modo 3 nos ofrece gráficos de baja resolución con 16 colores en la pantalla. En este modo también puedes usar *sprites*. A pesar de la presencia de color, en este modo no hay ninguna tabla de colores, sino que es la tabla de patrones la que tiene la información, tanto para la forma de la imagen como los colores que van a aparecer en la pantalla.

Tabla de nombres

La tabla de nombres en modo 3 ocupa 768 bytes y está en la dirección 2048 de la VRAM. Igual que la tabla de hombres en modo 2, ésta está también inicializada con determinados valores cuando introduces este modo de pantalla. Este proceso de inicialización lo veremos más adelante en este mismo capítulo. De todas formas, la tabla de nombres de este modo realiza la misma función que las otras

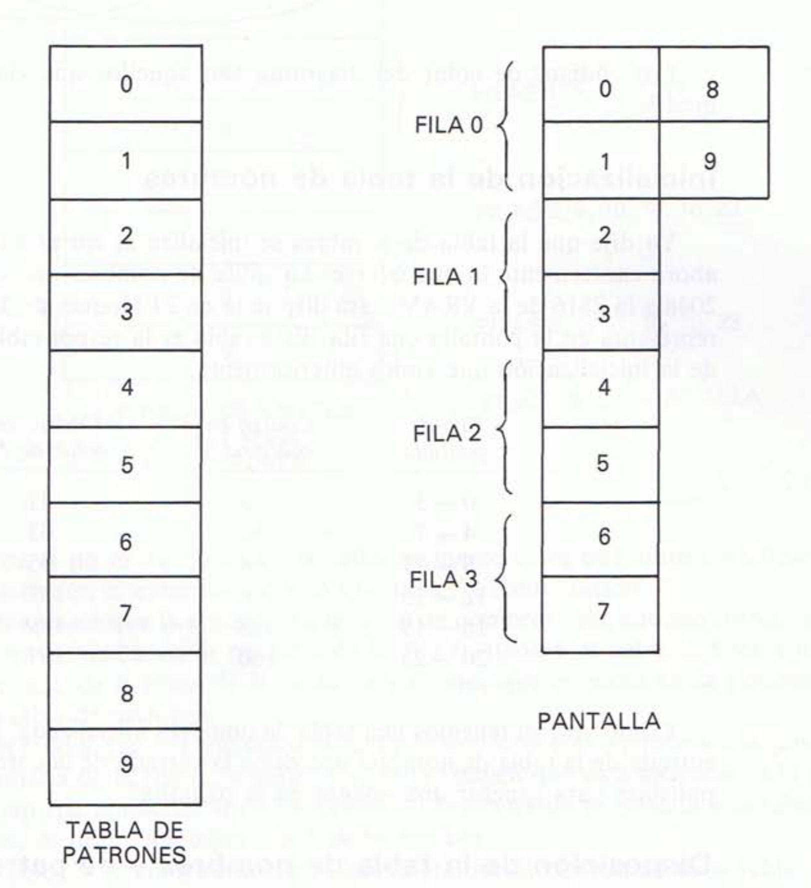
tablas de nombres en otros modos, es decir, sirven para indexar la tabla de patrones para definir la imagen en pantalla.

Tabla de patrones

Esta tabla de patrones ocupa 2048 bytes y está en la dirección 0 de la VRAM. Dicha tabla contiene la información sobre la imagen que va a aparecer en la pantalla y la información sobre los colores de las imágenes.

Los gráficos que nos ofrece el modo 3 no son de muy alta resolución; se puede decir que trabajamos con *superpixels* más que con los *pixels* comunes. Dichos *superpixels* son 4 por 4 *pixels* estándares.

La siguiente ilustración muestra cómo el proceso de inicialización de la tabla de nombres hace que la pantalla se sincronice con la tabla de patrones cuando entramos en el modo 3.

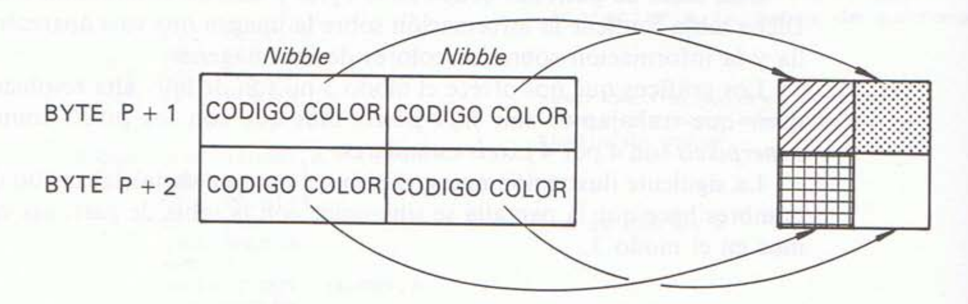


La tabla de patrones se sitúa en la pantalla en modo 3 después de la inicialización de la tabla de nombres.

Recuerda que la disposición anterior se debe a los valores descritos en la tabla de nombres durante el proceso de inicialización.

Cada byte de la tabla de modelos define dos *superpixels;* por tanto, cada carácter cuadrado en la pantalla necesita 2 bytes para definirlo, ya que está formado por 4 *superpixels*.

El siguiente diagrama muestra cómo se define dicho carácter:



Los códigos de color del diagrama son aquellos que vimos en los primeros modos.

Inicialización de la tabla de nombres

Ya dije que la tabla de nombres se inicializa al entrar en este modo; veamos ahora exactamente lo que ocurre. La tabla de nombres que va desde la dirección 2048 a la 2816 de la VRAM, está dispuesta en 24 bloques de 32 bytes; cada bloque representa en la pantalla una fila. Esta tabla es la responsable de la organización de la inicialización que vimos anteriormente.

Fila de pantalla	*Código en columna 0*	*Código en columna 31*
0 — 3	0	31
4 — 7	32	63
8 — 11	64	95
12 — 15	96	127
16 — 19	128	159
20 — 23	160	191

Como ves, ya tenemos una tabla de nombres inicializada, pero ¿cómo puede la entrada de la tabla de nombres acceder a la entrada de la correspondiente tabla de patrones para generar una imagen en la pantalla?

Disposición de la tabla de nombres y de patrones

Ya hemos visto cómo los ordenadores MSX inicializan la tabla de nombres. Seguidamente veremos la disposición de la tabla de patrones en este modo.

Ya hemos estudiado que se necesitan dos bytes para definir un carácter cuadrado en la pantalla; ahora bien, si cada entrada a la tabla de patrones ocupa 8 bytes, ¿qué pasa con los otros 8 bytes de la entrada a la tabla de patrones asociados con una determinada entrada a la tabla de nombres?

Pues bien, cada par de bytes en una entrada a la tabla de patrones define una imagen que puede salir en pantalla y cuyas entradas de dos bytes se usan para definir la imagen de una entrada de tabla de nombres determinada, dependiendo del lugar de la pantalla en donde va a salir la imagen. Las partes reales de la tabla de patrones utilizadas para una entrada de tabla de nombres determinada están en el diagrama siguiente:

ENTRADA A LA TABLA
DE PATRONES

BYTE	FILAS DE LA PANTALLA
0	FILAS 0, 4, 8, 12, 16, 20
1	
2	FILAS 1, 5, 9, 13, 17, 21
3	
4	FILAS 2, 6, 10, 14, 18, 22
5	
6	FILAS 3, 7, 11, 15, 19, 23
7	

ENTRADA DE 8 BYTES
A LA TABLA
DE PATRONES

Por si no te ha quedado lo suficientemente clara esta última explicación, lee con atención el ejemplo práctico que tienes a continuación:

Imagínate que la entrada 0 a la tabla de nombres, que además corresponde a la primera posición en la pantalla de la fila 0, tuviera el valor 3. Esta apuntará a la entrada de 8 bytes de la tabla de patrones que empieza en la posición $(3*8)$ de la tabla de patrones.

Debido a que esa entrada a la tabla de nombres corresponde a una posición en la pantalla de la fila 0, la definición de la imagen que va a aparecer en la pantalla será aquella almacenada en los primeros dos bytes de la entrada a la tabla de patrones, es decir, los bytes 0 y 1 de la entrada.

Vamos a ver ahora otro caso: imagínate que la entrada 34 de la tabla de nombres tuviera el valor 3, que corresponde a una posición de la fila 1 de la pantalla; de acuerdo con el diagrama anterior, los bytes 2 y 3 de la entrada a la tabla de patrones se usan para definir la imagen que va a salir en la pantalla.

También la entrada correspondiente a la fila 3 tomará la definición de los bytes 6 y 7 de la entrada y una entrada a la tabla de nombres en una posición de la tabla de nombres correspondiente a una posición en pantalla en la fila 4, tomará su definición, nuevamente, de los bytes 0 y 1 de la entrada de la tabla de patrones.

Escribir en estas dos tablas es un proceso bastante fácil, en el que puedes utilizar las mismas técnicas que ya has visto. Te aconsejo que uses, siempre que puedas, la tabla de patrones para modificar las imágenes que aparecen en la pantalla; esto no tiene por qué constituir problema alguno, siempre y cuando hayas entendido bien cómo se inicializa la tabla de nombres en este modo.

Los *sprites*

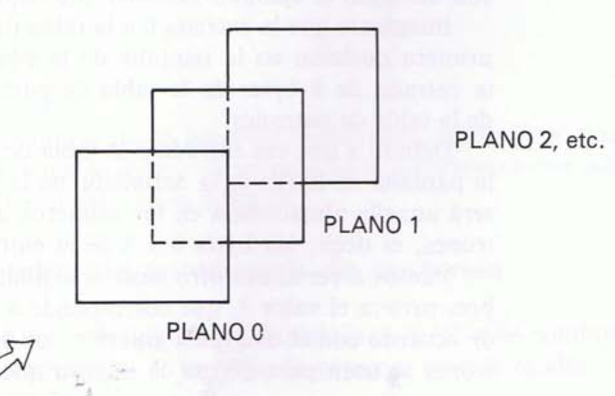
Bajo este epígrafe, he incluido una amplia explicación sobre el manejo de los *sprites,* en código máquina, para crear gráficos móviles con los ordenadores MSX; esta explicación va a ser bastante amplia, con objeto de que este importante tema quede muy claro y lo entiendas con la mayor rapidez y eficacia.

En esta sección, te voy a explicar cómo puedes definir y usar los *sprites;* además, voy a comentar algunos ejemplos conforme vayamos avanzando en el tema. Empezaré con una rápida explicación del sistema de *sprites* del MSX que espero te sea no sólo un tema de estudio sino también de entretenimiento, sobre todo para aquellos que no hayáis trabajado nunca con *sprites.*

Los *sprites* del MSX

Un *sprite* es, en términos muy simples, un carácter definible por el usuario que puede moverse por la pantalla simplemente especificando sus nuevas posiciones. Los *sprites* pueden pasar por delante de otros *sprites,* del texto o incluso por delante de gráficos.

Esta habilidad de los *sprites* proviene del hecho de que aparecen en la pantalla sobre lo que llamamos planos de visualización de los *sprites;* hay 32 planos de este tipo disponibles y están numerados del 0 al 31. El diagrama siguiente muestra la colocación de dichos planos en la pantalla del ordenador:

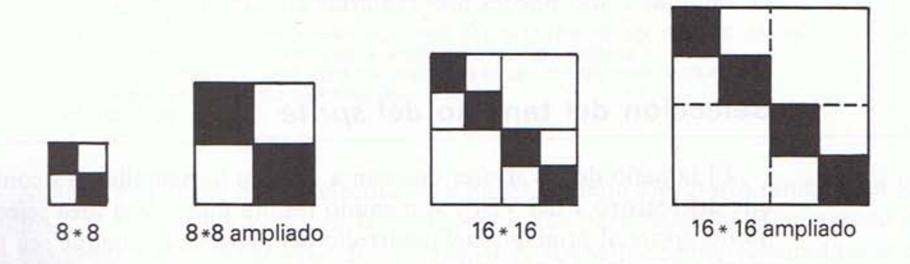


Cada plano puede contener un solo *sprite* a la vez. El *sprite* del plano 0, por ejemplo, pasará por delante de cualquier *sprite* de los otros planos cuando ambos *sprites* coincidan; por eso se dice que el *sprite* del plano 0 tiene mayor PRIORIDAD que cualquier otro *sprite;* así, el *sprite* del plano 1 tiene mayor prioridad que el del plano 2, etc. Detrás del *sprite* del plano 31 está el PLANO MULTICOLOR, que es el plano en el que aparecen los gráficos y el texto.

Como ves, cualquier tipo de plano de los 32 que te acabo de nombrar tendrá mayor prioridad que cualquier texto o gráfico.

En modo 0 no hay *sprites* disponibles, pero sí que los hay en los modos 1, 2 y 3. Los *sprites* pueden ser de diferentes tamaños: 8 * 8 *pixels,* 8 * 8 *pixels* ampliados, 16 * 16 *pixels* y 16 * 16 *pixels* ampliados.

Dichos tamaños tienen los siguientes diagramas:



El *sprite* de 8 * 8 *pixels* tiene el mismo tamaño que un carácter en los modos 1 y 2. El tamaño real de los *sprites* que van a utilizarse en los programas en BASIC se asigna con la instrucción SCREEN:

SCREEN modo, tamaño

donde "tamaño" es un entero entre 0 y 3. El tamaño para un determinado valor en el parámetro "tamaño" es:

Tamaño	*Tamaño del* sprite
0	8 * 8
1	8 * 8 AMPLIADO
2	16 * 16
3	16 * 16 AMPLIADO

La imagen que va a aparecer en la pantalla se almacena en la tabla de patrones de *sprite,* es decir, una zona de la VRAM que hay en cada uno de los tres modos de pantalla que admiten el *sprite.* Un *sprite* de 8 * 8 *pixels* necesita 8 bytes para ser almacenado dentro de la tabla anterior, y los *sprites* de 16 * 16 necesitan 32 bytes. La tabla de patrones de *sprite* ocupa 2048 bytes, y, por tanto, tiene lugar suficiente para almacenar 256 modelos de *sprites* distintos de 8 * 8 *pixels,* o, lo que es lo mismo, hay espacio para almacenar 64 *sprites* de 16 * 16 *pixels.* No se puede mezclar tamaños distintos de *sprites,* es decir, todos tienen que ser o normales o ampliados.

La posición de un *sprite* en la pantalla, su color y el *sprite* que se está usando son lo que se llama los ATRIBUTOS de un determinado plano de *sprites,* y se almacenan en una entrada de 4 bytes en la tabla con los atributos de los *sprites,* que es una zona de 128 bytes de la VRAM formada por 32 bloques de 4 bytes cada uno; cada bloque tiene la información de un determinado plano de *sprites.* Los primeros cuatro bytes tienen la información para el plano 0, incluido el número de patrones del *sprite* que está utilizando el plano anterior, su color y su posición.

El número del *sprite* que quieres se usa para indexar la tabla de patrones con la forma que va a salir en la pantalla; de forma parecida a como las entradas en la tabla de nombres en modos 1 y 0 indexan la tabla con los patrones para el cáracter que va a aparecer en pantalla.

Una vez terminada esta primera explicación sobre los *sprites* del MSX, es hora de estudiar cómo puedes programarlos en código máquina con tu ordenador.

Selección del tamaño del *sprite*

El tamaño de los *sprites* que van a salir en la pantalla está controlado por dos bits del registro 1 del VDP. A menudo resulta una buena idea seleccionar el tamaño del *sprite* al principio del desarrollo del programa, aunque sea posible pasar de *pixels* normales a los ampliados, y viceversa, una vez esté hecho el programa y sin ningún trastorno para los programas; pero si pasas del 8 * 8 al 16 * 16 se modificará la numeración de los patrones en la tabla con los patrones de *sprites* y pudiera acarrear un poco de lío.

El tamaño se controla con los bits 0 y 1 del registro 1. El efecto de las diferentes combinaciones de bits son:

	Bit 0	*Bit 1*
8 * 8	0	0
8 * 8 ampliado	0	1
16 * 16	1	0
16 * 16 ampliado	1	1

Como ves en esta tabla, el bit 1 controla la ampliación de los *sprites,* y el bit 0, el tamaño de los *sprites,* es decir, los *pixels* de los *sprites.*

En el ejemplo 35 veremos cómo asignar el tamaño de los *sprites* desde una rutina en código máquina.

EJEMPLO 35

En este ejemplo se activa el registro 1 del VDP para obtener *sprites* de 16 * 16 *pixels* en cualquier *sprite* del tipo de pantalla. Fíjate en cómo se usan las copias de la RAM del registro del VDP apropiado para que no afecten a ningún bit, excepto los que tú elijas.

```
        IN     A,(&H99)              ;borra el VDP
        LD     A,(&HF3E0)            ;consigue de la RAM la copia
                                     ;del registro 1 deL VDP
        SET    0,A                   ;asigna el bit 0 y el 1 de a-
                                     ;cuerdo con la tabla anterior
        SET    1,A
        LD     (&HF3E0),A            ;actualiza la copia de la RAM
        OUT    (&H99),A
        LD     A,&H81
        OUT    (&H99),A
        RET
```

El siguiente programa en BASIC utiliza la rutina anterior.

```
10 SCREEN 1,2: REM INICIALIZA LOS SPRITES A 16x16 PIXELS
20 DEFUSR=59000: REM DIRECCION DE LA RUTINA
30 A$= "abcdefgh"
40 SPRITE$ (1)= A$+A$+A$+A$ : REM DEFINICION DE UN SPRITE 16x16
50 PUT SPRITE(1),(100,100): REM MUESTRA EL SPRITE
60 L$=INPUT$(1): REM ESPERA HASTA QUE SE PULSA UNA TECLA
70 L=USR(0); REM EJECUCION DEL CODIGO
80 GOTO 80
```

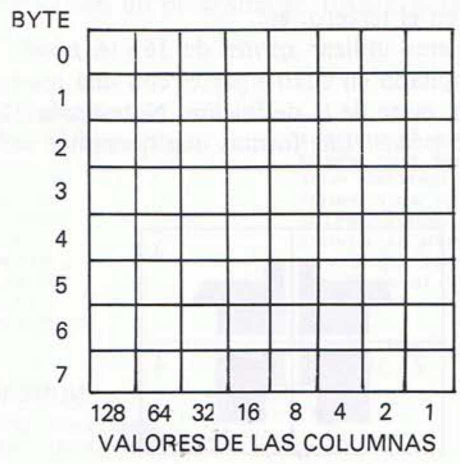
Aunque este programa se ejecuta en modo 1, intenta ejecutarlo también en los modos 2 y 3 para que veas que la técnica que uso para recobrar el contenido en ese momento del registro VDP de la copia de la RAM funciona. Recuerda que en los modos de gráficos tienes que poner una línea 80 como la siguiente:

80 GOTO 80

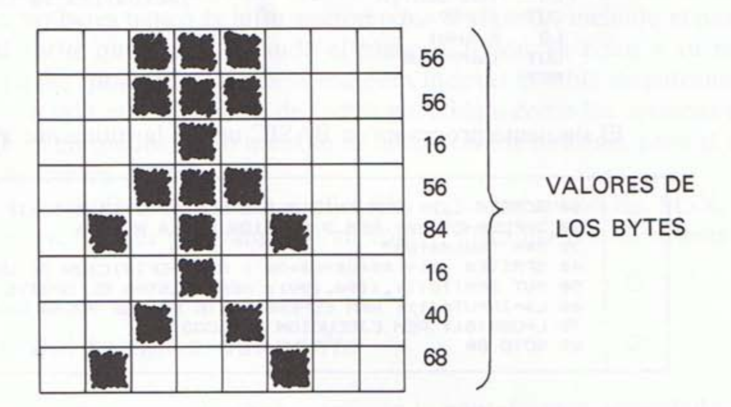
para prevenir posibles retornos accidentales al modo de texto antes de lo previsto.

Cómo definir un *sprite*

El principio fundamental para definir un *sprite* reside en el uso de una matriz 8 ∗ 8 parecida a un tablero de ajedrez sobre el cual dibujamos la imagen que nos gustaría para un número de patrón de *sprite* determinado. La matriz sería como la siguiente:



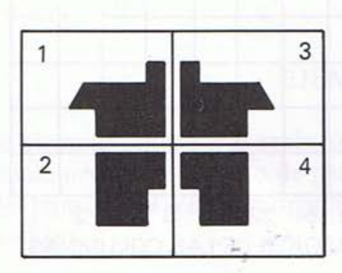
Para definir un *sprite* de 8 * 8 *pixels,* simplemente coloreamos los cuadrados de la matriz que vayan a formar la figura. Por ejemplo, para definir un *sprite* que represente un pequeño hombre nos quedaría algo parecido a la matriz siguiente.



Los valores de los bytes se obtienen sumando todos los valores de las columnas de los cuadrados sombreados en cada línea horizontal de la matriz. ¿Pero qué hacemos con estos valores? Pues bien, en BASIC usamos la instrucción SPRITE$ () para escribir esos valores en la tabla de patrones de *sprite,* pero en código máquina tenemos que escribirlos directamente en la tabla de la misma forma en que escribíamos nuevas definiciones de caracteres en modo 1 y modo 0 en las tablas de patrones.

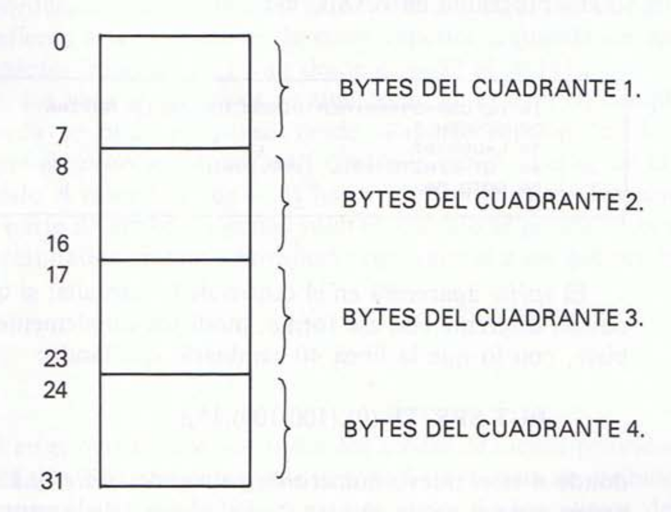
La tabla generadora de patrones de *sprites* (para darle su título completo) está situada en la dirección 14336 de la RAM, o lo que es lo mismo, &H3800; de esta manera los primeros 8 bytes tendrán la definición para el patrón de *sprite* considerando que estamos usando 8 * 8 *pixels.* En el ejemplo anterior habíamos escrito el valor 56 en el primer byte de la tabla con los patrones de *sprites,* 56 en el segundo byte, 16 en el tercero, etc.

Si quieres utilizar *sprites* de 16 * 16 *pixels,* se hace igual pero definiendo el *sprite* ampliado en cuatro partes con una matriz como la que vamos a ver ahora para cada parte de la definición. Necesitarás 32 bytes para almacenar cada *sprite* de 16 * 16 *pixels.* Las formas que queremos se dibujan en cuatro matrices como sigue:



NUMERO DEL CUADRANTE

Estos valores se escribirán después en la entrada en la tabla de patrones de la siguiente manera, considerando que estés definiendo el patrón de *sprite* 0.



Por ejemplo, para un *sprite* de 16 * 16 como el patrón número 0, los primeros 8 bytes de la definición definirán el cuadrante 2; los siguientes 8 bytes el segundo cuadrante, etc. La definición para el *sprite* con el patrón número 1, por tanto, empezará en la posición 32 de la tabla de patrones de *sprites,* etc.

Pasemos ahora a ver una rutina en código máquina que usamos para definir un *sprite* de 8 * 8.

EJEMPLO 36

Esta rutina define al patrón de *sprite* 0 como nuestro pequeño hombre. El listado es el siguiente, junto con un programa en BASIC para verlo mejor.

```
        DI
        IN   A,(&H99)
        LD   HL,TABLA               ;asigna el par de registros HL
                                    ;para que señale a los bytes
                                    ;que definen el caracter
        LD   B,&H08                 ;numero de bytes de datos
        LD   C,&H98                 ;direcciones de E/S para OTIR
        LD   A,0                    ;envia la direccion del primer
        LD   (&H99),A               ;byte de la tabla patron de
        LD   A,&H38                 ;sprites al VDP
        ADD  A,64
        OUT  (&H99),A
        OTIR                        ;manda los datos para la defi-
                                    ;nicion
        EI
        RET
TABLA   &H38,&H38,&H10,&H38
        &H54,&H10,&H28,&H44
```

Los bytes para los datos introdúcelos como una cadena de bytes. Asegúrate de que tienes la dirección de la tabla correcta; de otra manera, la imagen del *sprite* que obtendrás será más bien rara.

El programa en BASIC es:

```
10 DEFUSR=59000:REM DIRECCION DE LA RUTINA
20 SCREEN 2,2
30 L=USR(0)
40 PUT SPRITE (0), (100,100)
50 GOTO 50
```

El *sprite* aparecerá en el centro de la pantalla; si quieres definir un número de patrón diferente con esa forma, modifica simplemente la dirección en la que escribiste, con lo que la línea 40 cambiará, quedando:

PUT SPRITE (0),(100,100),15,n

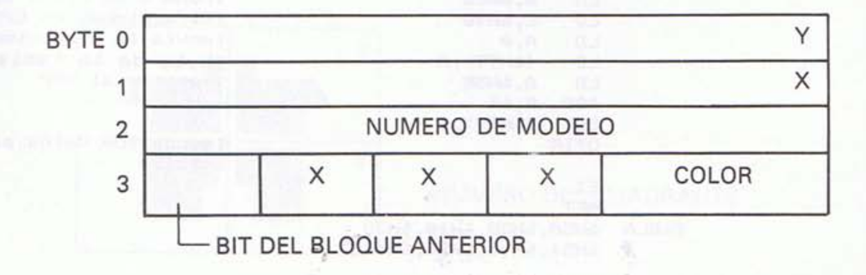
donde *n* es el nuevo número de patrón de *sprite,* y 15 es el código del color. Esta orden pone el *sprite* número 0 en el plano 0 de la pantalla en vez de poner el número 0 del patrón del *sprite* en el plano 0, que sería lo normal. Para más información sobre cómo manejar *sprites* en BASIC dirígete a tu manual del MSX.

Ahora ya podemos pasar a definir los patrones de *sprite* y decidir si queremos que éstos sean normales o ampliados. Veamos cómo se asignan y mueven los *sprites* en la pantalla del MSX.

Cómo mover y asignar los *sprites*

Tanto la posición como el color y el número de patrón de un *sprite* que vaya a aparecer en cualquiera de los 32 planos de la pantalla se almacenan en la entrada a la tabla de atributos de los *sprites,* correspondiente a ese plano de la pantalla. Hay 32 entradas, cada una de 4 bytes, dando un total de 128 bytes, que es lo que ocupa esa tabla.

La tabla de atributos de los *sprites* empieza en la dirección &H1B00 de la VRAM, es decir, en la 6912. En el siguiente diagrama vemos cómo se dispone un elemento de tabla de atributos de los *sprites.*



El byte 0

Este byte tiene la posición Y del *sprite* en el plano al que pertenece la entrada a la tabla. Las coordenadas X e Y situadas en una entrada en la tabla de atributos de los *sprites* se refieren a la posición en la parte superior izquierda del *sprite* en la pantalla. Los valores útiles de la Y van desde el —32 al +191.

Obviamente, los valores negativos se almacenan en el byte en complemento a dos. La ordenada se mide en *pixels* desde la parte superior de la pantalla; un —1 haría que el *sprite* se "aplastara" contra la parte superior de la pantalla.

Si vas variando el valor Y desde —31 hasta 0, el *sprite* saldrá lentamente en la pantalla por la parte de arriba. Algunos valores, cuando se ponen en la entrada Y de la zona dan resultados un tanto extraños; estos valores a los que me refiero son el 208 y el 209.

Y = 208

El valor 208 en el byte 0 hace que todos los *sprites* de menor prioridad desaparezcan de la pantalla. Es decir, si asignas al byte 0 de la zona de atributos para el plano 0 de la pantalla el valor 208, cualquier *sprite* de los planos del 1 al 31 desaparecerá. Esta situación permanece inalterable hasta que ocurra otra entrada.

Y = 209

El valor 209 en el byte 0 hace desaparecer el *sprite* que se encuentre en el plano que la zona de atributos indique. Por ejemplo, si asignas el valor 209 al byte 0 de la pantalla, los *sprites* del plano 0 de la pantalla desaparecerían y dejarían como estaban los *sprites* de los otros planos.

El byte 1

Este byte tiene la coordenada X en el plano de la pantalla del *sprite*. Los valores que puedes usar van desde el 0 al 255. La posición X del *sprite* se mide desde el margen izquierdo de la pantalla, en *pixels*.

Fíjate que tanto el movimiento como la asignación de *sprites* del MSX en una matriz son los mismos que en la resolución del modo de pantalla 2.

El byte 2

Este byte tiene el número del modelo de *sprite* que va a aparecer en el plano de la pantalla. Por ejemplo, si asignas 32 al byte 2 de la zona de atributos correspondiente al plano 0, aparecerá el modelo de *sprite* 32 en el plano de la pantalla 0.

Cuando usamos *sprites* en el BASIC, este byte se activa para que tenga el nú-

mero del plano al que corresponda la zona de atributos. De esta manera la zona de atributos del plano 0 de la pantalla tendrá ese byte a 0, el que corresponda a 1 tendrá su byte 2 a 1 y así sucesivamente.

El byte 3

La función de este byte en la zona de atributos es doble.

El color

Los cuatro bits menores de este byte tienen la información sobre los colores del *sprite* que van a aparecer en ese plano. Estos cuatro bits se usan para almacenar el código del color del fondo de los *sprites*.

Los códigos de color usados son los mismos que aquellos que ya vimos en los modos de pantallas. Los *pixels* de un *sprite* que no están en los colores de fondo son transparentes, para que puedas ver lo que hay en planos de pantalla inferiores.

Los bits 4 y 6

Normalmente estos bits no se usan.

El bit 7

A este bit se le llama bit *EARLY CLOCK BIT*. Este nombre deriva del lenguaje del *hardware* del ordenador. Los programadores sólo necesitamos recordar lo que hace más que lo que este nombre significa.

Si se fija a 0, no aparece ningún efecto, pero cuando se activa el *sprite* del plano en cuestión se asigna en X-32,Y; en vez de hacerlo en el X,Y normal. Así, lo que ocurre al asignar un 1 a este bit es que el *sprite* se mueve 32 unidades hacia la izquierda.

Con esto doy por terminada la teoría sobre el control de *sprites;* pasemos ahora a ver algunos ejemplos prácticos con *sprites*.

El ejemplo 37 muestra cómo asignar una zona de atributos de pantalla muy simple con una rutina en código máquina.

EJEMPLO 37

Este listado asigna una zona de parámetro al plano 0 de la pantalla del *sprite*. Tenemos, pues, que escribir información en los cuatro primeros bytes de la tabla de atributos.

```
DI
IN   A,(&H99)
LD   HL,58000                  ;coge informacion de la RAM y
LD   C,&H98                    ;la envia al VDP
LD   B,&H04
LD   A,&H00
OUT  (&H99),A                  ;envia la direccion del primer
LD   A,&H1B                    ;byte de la zona apropiada de
ADD  64                        ;la tabla de atributos
OUT  (&H99),A
OTIR                           ;envia los datos de la copia de
EI                             ;la RAM
RET
```

La rutina anterior puede probarse con el siguiente programa en BASIC. Escribe el código anterior en la dirección 59000.

```
10 SCREEN 1,1
20 DEFUSR=59000
30 SPRITE$(0)="abcdefg" : REM U OTRA DEFINICION
40 POKE 58000,100:REM POSICION Y DEL SPRITE
50 POKE 58001,200: REM NUMERO DEL SPRITE
60 POKE 58002,2: REM NUMERO DEL SPRITE
70 POKE 58003,15: REM NUMERO DEL SPRITE
80 L= USR(0)
```

Las cuatro instrucciones POKE simplemente crean información para que el programa en código máquina la transfiera a la VRAM. El *sprite* aparecerá en el centro de la pantalla. Para cambiar cualquier valor en la entrada de la tabla de atributos simplemente cambia los valores que, con la orden POKE, se asignaron en posición correspondiente de la RAM. Después llama a la rutina USR para actualizar la entrada en la tabla de atributos.

Mover un *sprite* por la pantalla es bastante fácil. Simplemente modifica el contenido de los bytes 0 y 1 de la zona de atributos apropiada. Por ejemplo, para mover un *sprite* del plano 1, sólo tienes que modificar los bytes en el segundo grupo de 4 bytes de la tabla con los atributos.

El ejemplo 38 es una rutina que mueve un *sprite* de izquierda a derecha de la pantalla a una velocidad sorprendente; se recuerda especialmente que la siguiente rutina tiene un bucle de retardo para dicha velocidad.

EJEMPLO 38

El listado es el siguiente:

```
DI
IN   A,(&H99)
LD   HL,58000
LD   C,&H98
LD   B,&H04
LD   A,&H00
OUT  (&H99),A
LD   A,&H1B
ADD  64
```

```
        OUT   (&H99),A
        OTIR                                ;envia los atributos iniciales
                                            ;para el plano 0
        LD    C,&H00
        LD    B,&HFF
 BUCLE  LD    A,&H01                        ;envia la direccion de la coor-
                                            ;denada X
        OUT   (&H99),A                      ;entrada al atributo del
                                            ;plano 0
        LD    A,&H1B                        ;bloque
        ADD   64
        OUT   (&H99),A
        LD    A,C                           ;coge la nueva coordenada X
        OUT   (&H98),A                      ;desde el  registro C y  la
                                            ;manda
        INC   C                             ;lo deja para la siguiente vez
        LD    HL,512                        ;ejecuta un bucle de retardo
RETARD  DEC   HL
        LD    A,H
        OR    L
        JR    NZ,RETARD
        DJNZ  BUCLE
        EI
        RET
```

El programa en BASIC que demuestre lo anterior:

```
10 SCREEN 1,1
20 DEFUSR=59000:M$= ""
30 FOR I=1 TO 8
40 READ N:N$=CHR$(N)
50 M$=M$+N$
60 NEXT I
70 SPRITE$(0)=M$
80 POKE 58000,100
90 POKE 58001,0
100 POKE 58003,0
110 POKE 58003,1
120 L=USR(0)
130 DATA 56,56,16,56,84,16,56,84,16,40,68
```

Rápido, ¿verdad? Si quitases el bucle de retardo, el *sprite* se movería tan rápido que ni siquiera lo podrías ver. La velocidad es la misma que para los *sprites* de 16 ∗ 16 *pixels,* ya que lo más pesado lo hace el VDP.

Como es lógico, también puedes mover *sprites* con el teclado al igual que los puedes mover programando dicho movimiento, como acabamos de hacer. Para mover los *sprites* con tu teclado, simplemente modifica los bytes necesarios de la entrada de la zona de atributos para el plano en el que está el sprite, de acuerdo con las teclas que vayas a pulsar.

EJEMPLO 39

En esta rutina veremos cómo con el teclado podemos mover un *sprite* a la derecha y a la izquierda. El sentido del movimiento depende de si pulsas la tecla que mueve el cursor a la derecha o si pulsas la que lo mueve a la izquierda. Para volver al BASIC pulsa la tecla DELETE. Como ya vimos, aquí también usamos el pla-

no 0 del *sprite,* por lo que el primer bit del programa simplemente asigna la zona de atributos de la pantalla para ese plano.

```
        DI
        IN   A,(&H99)
        LD   HL,58000
        LD   C,&H98
        LD   B,&H04
        LD   &H00
        OUT  (&H99),A
        LD   A,&H1B
        ADD  64
        OUT  (&H99),A
        OTIR
        LD   C,&H00
BUCLE1  LD   A,1              ;escribe el valor X en la zo-
        OUT  (&H99),A         ;na de atributos
        LD   A,&H1B
        ADD  64
        OUT  (&H99),A
        LD   A,C
        OUT  (&H98),A         ;envia el valor
        LD   HL,256           ;fija un bucle de retardo
RETARD  DEC  HL
        LD   A,H
        OR   L
        JR   NZ,RETARD        ;ejecuta el retardo
        LD   A,&HF8           ;lee el teclado
        OUT  (&HAA),A
        IN   A,(&HA9)
        BIT  3,A              ;comprueba la tecla DELETE
        JR   Z,SALIDA
        BIT  4,A
        JR   NZ,TEST2         ;comprueba las teclas
                              ;del  cursor
        DEC  C
        JR   BUCLE
TEST2   BIT  7,A
        JR   NZ,BUCLE
        INC  C
        JR   BUCLE
SALIDA  EI
        RET
```

Aquí también usamos inicialmente un programa en BASIC para asignar la zona de atributos.

```
10 SCREEN 1,1
20 SPRITE$(0)="abcdefgh":REM O LA DEFINICION DE TU SPRITE
30 DEFUSR=59000
40 POKE 58000,100
50 POKE 58001,0
60 POKE 58002,0
70 POKE 58003,1
80 L=USR(0)
```

Si no escribes el bucle de retardo, no podrás ver lo que ocurre adecuadamente, ya que la velocidad del *sprite* sería demasiado rápida para poder verlo bien.

Lo último que vamos a ver referente a los *sprites* del MSX es el *flag* de coincidencia de los *sprites.*

Coincidencia de *sprites*

Una coincidencia de *sprites* se detecta comprobando el estado de un determinado bit en el registro de estado del VDP llamado *flag* de coincidencia que es el bit 5 del registro de estado y se le asigna un 1 siempre que dos *sprites* coinciden. Incluso cuando sólo coinciden dos *pixels,* uno por cada *sprite,* este *flag* se activaría.

El *flag* se pone a 0 después de ejecutar una operación de lectura del registro de estado.

Sin embargo, existen varios problemas cuando leemos este *flag* desde el código máquina:

1. El estado del *flag* de coincidencia se actualiza cada dos centésimas de segundo. Toda la pantalla examina el VDP para localizar dos *pixels* superpuestos. Pero si estuvieras moviendo muy deprisa los *sprites* o si los estuvieras moviendo con diferentes *pixels* a la vez, es posible que el VDP ignore una superposición que ocurra demasiado deprisa para detectarla.
2. La interrupción que genera el VDP cada dos centésimas de segundo hace que la UCP lea el registro de estado del VDP, con lo cual, por ejemplo, borraría el *flag* de coincidencia de *sprites* cada dos centésimas de segundo; así cualquier coincidencia entre dos interrupciones tendría que ser detectada leyendo el registro de estado del VDP DESPUES que haya ocurrido una coincidencia y ANTES de que la interrupción haga que se lea el registro de estado.
3. El último problema es que, si leemos el registro de estado entre las interrupciones, cualquier interrupción que esté esperando a ser ejecutada por el VDP se borrará.

Las dos últimas complicaciones, que no deberían ser tales, pueden producirse al desactivar las interrupciones cuando introduzcas un programa de gráficos en código máquina en el que tengas que examinar la coincidencia de *sprites;* pero, como ya te has enfrentado con esta posible dificultad, espero que no te cause ningún problema.

En el ejemplo 40 tenemos un ejemplo de cómo examinar con una rutina en código máquina la coincidencia de *sprites.*

EJEMPLO 40

Empecemos con el listado para a continuación pasar a la explicación y el programa BASIC de demostración.

```
IN1     DI                             ;la primera entrada seØala USR0(0)
        LD     HL,58000
        LD     C,&H98
        LD     B,&H04
        LD     A,&H00
        OUT    (&H99),A
        LD     A,&H1B
```

```
        ADD   64
        OUT   (&H99),A
        OTIR
IN2     DI                          ;la segunda entrada señala USR1(0)
        LD    HL,&H0500             ;bucle de retardo que espera hasta
                                    ;que se compruebe una coincidencia
RETARD  DEC   HL                    ;de dos sprites
        LD    A,L
        OR    H
        JR    NZ,RETARD
        IN    A,(&H99)              ;lee el registro de estado
        BIT   5,A                   ;comprueba el estado del flag de
                                    ;coincidencia de sprites
        JR    Z,SALIDA
        LD    A,1
        LD    (&HF7F8),A
        LD    A,2
SALIDA  LD    (&HF663),A
        EI
        RET
```

Fíjate que hay dos puntos que se llaman por funciones USR. La primera, IN1, se llama para inicializar el proceso, y la segunda, IN2, para comprobar la coincidencia de *sprites* cuando sea necesario. La dirección de comienzo de la rutina, suponiendo que escribes el código en la dirección 59000, será la 59022. El programa demostración en BASIC es el que tienes a continuación.

```
10 SCREEN 1,1
20 DEFUSR0=59000
30 DEFUSR1=59022
40 POKE 58000,100
50 POKE 58001,100
60 POKE 58002,0
70 POKE 58003,1
80 SPRITE$(0)=def$:REM def$=definicion de sprite
90 SPRITE$(1)=def$:REM como en la linea anterior
100 PRINT USR(0)
110 FOR Y=80 TO 140
120 PUT SPRITE(1),(100,Y),15
130 FOR I=0 TO 1000:NEXT:REM retraso para leer los numeros
escritos
140 PRINT USR1(0)
150 NEXT
```

Ejecuta el programa y fíjate que la rutina que se llama en la línea 140 escribirá un número en la pantalla, siempre que uno o más *pixels* de los *sprites* coincidan.

Flag del quinto *sprite*

Este *flag* indica al usuario que se ha intentado asignar cinco *sprites* en la misma línea horizontal. Este *flag* se activa siempre que eso ocurre, y el número del plano que tiene el quinto *sprite* se asigna en los 5 bits menores del registro de estado del VDP. Dichos bits sólo darán validez a un número de plano cuando el quinto *flag* del *sprite* esté activado.

El *flag* se borra leyendo el registro de estado; de esta forma se te avisa de que sigues el método anterior para comprobar correctamente este *flag*.

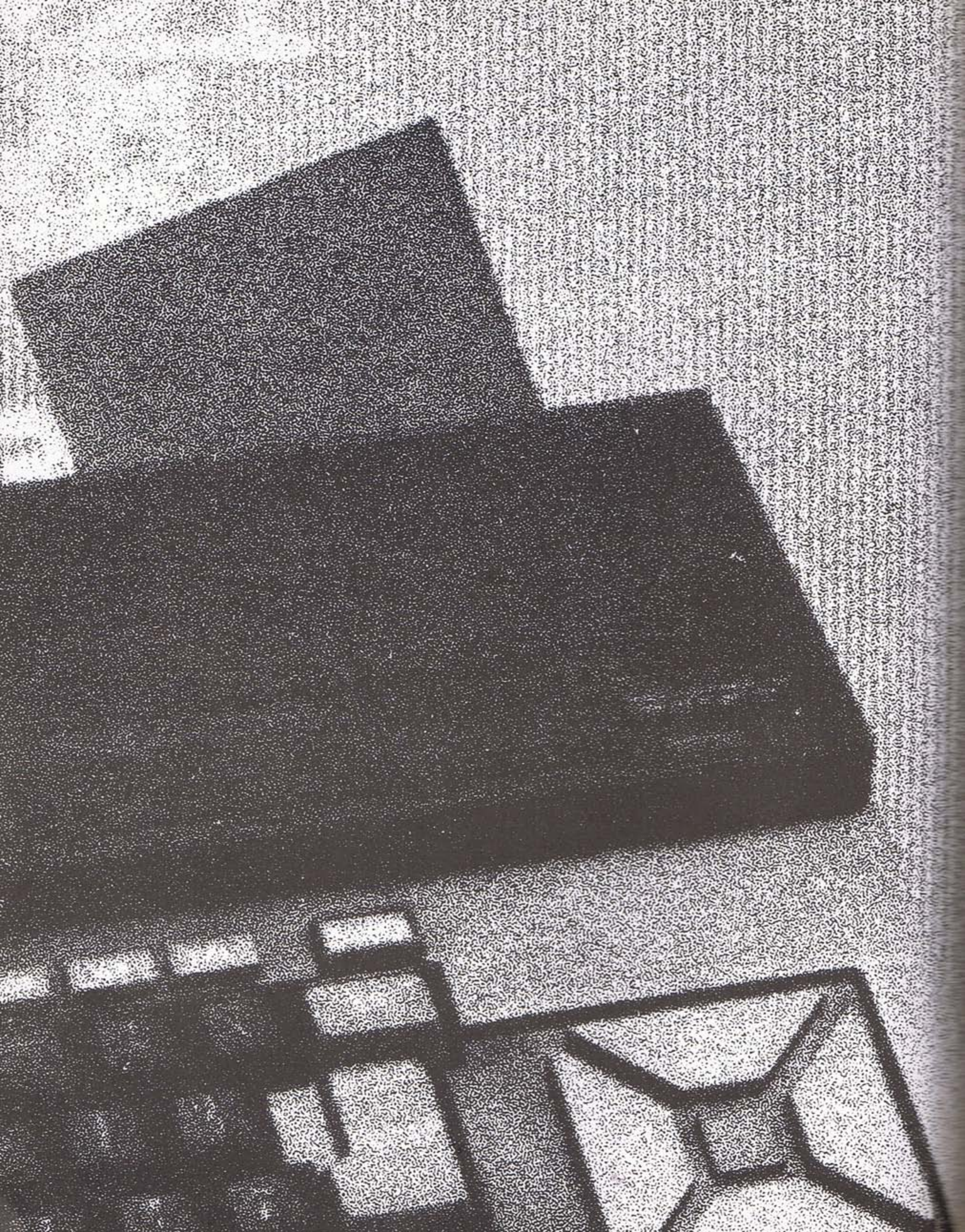
Por último, recuerda que tanto el quinto *flag* del *sprite* como el de coincidencia se borran al leer el contrario, es decir, el quinto *flag* del *sprite* se borrará leyendo el de coincidencia, y viceversa; por tanto, es aconsejable leer el registro VDP y almacenarlo en la RAM para futuras referencias, actualizando la copia de la RAM cada vez que se lea el registro de estado.

Con esto queda completa esta guía de programación del VDP en código máquina para principiantes; seguro que hay en este capítulo suficiente información para que puedas dibujar líneas y rutinas que pinten puntos cuando la pantalla está en modo gráfico.

Este último aspecto se sale un poco del propósito inicial que persigue este libro, orientado principalmente para principiantes, así que no me voy a extender más en este tema.

En el capítulo siguiente veremos cómo programar el generador de sonido programable o PSG para así poder unir a los gráficos que acabas de ''descubrir'' en código máquina con sonidos también programados en código máquina.





11

El generador programable de sonido (PSG)

El PSG, sigla que corresponde a los términos ingleses *Programmable Sound Generator* (Generador Programable de Sonido, en español), es un dispositivo del sistema MSX es el responsable de generar la amplia gama de sonidos que el ordenador es capaz de crear.

Ya vimos en el capítulo 9 cómo generar un sonido simplemente encendiendo y apagando el sistema de sonido del MSX a través de los registros PPI. Sin embargo, este método para generar sonido tiene importantes limitaciones que lo hacen inútil a la hora de generar sonidos, digamos, más complicados; la PPI requiere total atención por parte de la UCP para producir sonido, incluso cuando desactiva la interrupción, lo cual significa, obviamente, que, mientras la UCP sea usada para generar sonidos, no podrá hacer absolutamente nada más.

La solución a este problema es utilizar un dispositivo cuya sola tarea sea la de producir sonidos con el mínimo de intervención de la UCP del ordenador. En los ordenadores MSX este dispositivo se llama AY-3-8910 PSG y lo único que tiene que hacer la UCP es informar al PSG sobre el tipo de sonido que tiene que producir, cuándo tiene que empezar a generar sonido y cuándo tiene que parar.

El PSG posee todos los mecanismos para producir sonidos sin necesidad de intervención por parte de la UCP.

Conceptos generales sobre el AY-3-8910

El 8910 es un dispositivo de tres canales, es decir, puede generar tres tonos simultáneamente y cada tono con diferente grado y volumen que los otros dos, en caso de que así lo necesites.

Cada canal puede producir ruido, con lo que este dispositivo es muy aconsejable utilizarlo para crear efectos especiales con los sonidos. Una serie de registros se encargan, al igual que en el VDP, de controlar el PSG. Hay también dos registros de ENTRADA/SALIDA que se usan, en el MSX, para controlar los mandos de juego o *joysticks.*

Al 8910 se accede a través de tres posiciones del mapa de entrada/salida del MSX y así puedes escribir en él con las instrucciones IN y OUT del Z-80.

Mientras que los registros del VDP se dividían en registros sólo para leer o sólo para escribir, los registros del PSG son todos para leer y escribir, es decir, como si fueran posiciones típicas de la RAM; lo cual significa que el usuario puede encontrar el valor en ese momento de cualquier registro PSG.

Cómo acceder al PSG

Al PSG se accede, como ya te he dicho, a través de tres posiciones del plano de entrada/salida del MSX.

&HA0

Esta se llama DIRECCION DE ACCESO del generador programable de sonido y tiene el número del registro PSG al que quieres acceder, por lo cual tendrás que escribir el número del registro elegido en esa posición antes de pasar a acceder al registro.

El valor que podemos escribir en esa posición va desde el 0 al 15.

&HA1

Es la posición para escribir datos en el PSG; una vez que hayas asignado un número de un registro en &HA0, una operación de escritura escribirá el valor de esa posición en el registro cuyo número esté en la posición &HA0.

&HA2

Es la posición para leer datos del PSG y opera de forma parecida a como lo hace la posición &HA1. Una vez que una dirección correcta esté en una posición &HA0, una instrucción IN desde esa misma posición transferirá el contenido del

registro PSG, cuyo número está en el &HA0, al registro de la UCP implicado en la operación IN.

Los registros del PSG

En el PSG hay dieciséis registros, de los cuales catorce tienen que ver con la generación de sonidos. Los otros dos, que veremos al final del capítulo, se llaman registros paralelos de entrada/salida, y su funcionamiento es muy parecido al de los tres registros de entrada/salida del interfaz programable de periféricos. Se usan para conectar la UCP con los mandos de juegos.

Veamos ahora cómo escribir y leer información en los registros PSG.

Cómo escribir en los registros del PSG

Simplemente es una cuestión de saber escribir el número del registro y enviar la información de la siguiente manera:

```
LD     A,8         ; número del registro
OUT    (&HA0),A    ; mándalo al PSG
LD     A,15        ; información para el registro
OUT    (&HA1),A    ; mándalo al PSG
RET
```

Este ejemplo escribe el valor 15 en el registro 8 del PSG. Más adelante veremos más detalladamente lo que hace cada registro.

Cómo leer los registros del PSG

Leer los registros del PSG es muy fácil: sencillamente tienes que enviar el número del registro a la dirección de acceso y después ejecutar una instrucción IN desde la dirección en la cual se va a leer la información:

```
LD     A,8         ; manda el registro
OUT    (&HA0),A    ; al PSG
IN     A,(&HA2)    ; lee el valor devuelto
LD     (59000),A
RET
```

Esta sencilla rutina almacena el contenido del octavo registro PSG en la dirección 59000 de la RAM. A diferencia del registro de estado del VDP, leer un registro del PSG deja su contenido sin modificar.

Una vez que hayas aprendido cómo acceder a estos registros será hora de pasar a ver qué hacen cada uno de los registros del PSG.

Descripción de los registros del PSG

A los registros del PSG se puede acceder desde el BASIC usando las instrucciones INP y OUT o la instrucción SOUND:

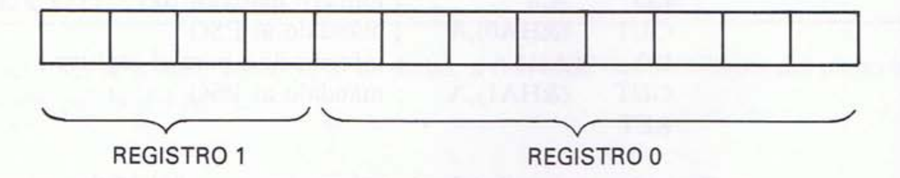
SOUND registro,datos

escribirá el valor que haya en "datos" en el registro con el número que haya en "registro". La única forma de leer el contenido de un registro del PSG desde el BASIC es usando la instrucción INP.

Pasemos a estudiar ahora cada registro.

Los registros 0 y 1

El PSG trabaja con ellos simultáneamente; estos registros tienen un número de 12 bits. Los 8 bits bajos del registro están formados por el contenido del registro 0, y los 4 bits altos del número están formados por los 4 bits bajos del registro 1. Con el dibujo siguiente lo entenderás mejor:



Los bits altos del registro 1 no se usan. El número de 12 bits controla el grado del sonido que sale del canal 1 del PSG. Es lógico pensar, pues, que el contenido del registro 1, que es el más significativo con respecto al número de 12 bits, tendrá un efecto mayor en el grado de la nota que el contenido del registro 0. Por esta razón, al registro 1 se le suele llamar REGISTRO DE CONTROL DE GRAVES DEL TONO y al registro 0 se le llama REGISTRO DE CONTROL DE AGUDOS DEL TONO.

Cuanto mayor sea el valor que tenga el registro de 12 bits, menor será el grado del tono generado por el canal 1. Así, para escribir un valor en este par de registros tendrás que usar una rutina como la siguiente:

```
LD     A,0
OUT    (&HA0),A    ; selecciona el registro número 0
LD     A,&FF
OUT    (&HA1),A    ; manda el dato al registro 0
LD     A,&H01
OUT    (&HA0),A    ; selecciona el registro 1
LD     A,0
OUT    (&HA1),A    ; manda el dato al registro 1
RET
```

Los registros 2 y 3

Estos dos registros realizan una función parecida a los registros 0 y 1, pero con la diferencia de que éstos operan sobre el canal 2 del PSG. El registro 2 es el registro de control de agudos del tono, y el registro 3 es el registro de control de graves del tono.

Los registros 4 y 5

Estos registros también se organizan, en líneas generales, como lo hacían los registros 0 y 1, pero éstos controlan el grado del tono que genera el canal 3 del PSG. El registro 4 es el registro de control de agudos del tono, y el registro 5 es el registro de control de graves del tono.

Por el momento pasaré por alto los registros 6 y 7 para pasar a estudiar otros registros que están implicados principalmente en la generación de tonos puros.

No creas que simplemente asignando el grado de un tono en un determinado canal provoca inmediatamente que se genere ese tono en el canal apropiado; al igual que el grado, también tienes que especificar la amplitud del tono.

Control de amplitud

La amplitud o volumen de una nota determina su intensidad. Hay un registro de control de amplitud para cada canal del PSG y todos son de 4 bits; lo cual nos da dieciséis niveles diferentes de volumen: desde 0, que es silencio, hasta el 15, que es la intensidad máxima.

El registro 8

Este es el registro de control de amplitud para el canal 1. Por el momento será considerado como un registro de 4 bits, aunque el quinto bit lo use el PSG, como veremos más adelante.

El registro 9

Es el registro de control de amplitud para el canal 2.

El registro 10

Es el registro de control de amplitud para el canal 3.

Creo que ya debes saber lo suficiente sobre el generador programable de sonido como para ser capaz de generar algún sonido sencillo con el PSG.

EJEMPLO 41

Este programa genera un sonido por el canal 1 durante un período muy corto de tiempo. Aquí tienes la rutina y a continuación la explicación:

```
          DI
          LD   A,0                    ;asigna el grado de control
          OUT  (&HA0),A               ; del  registro  para  el
          LD   A,&HFF                 ;canal 1
          OUT  (&HA1),A
          LD   A,1
          OUT  (&HA0),A
          LD   A,8
          OUT  (&HA1),A
          LD   A,8                    ; manda el numero de regis-
          OUT  (&HA0),A               ;tro de control de amplitud
          LD   A,8                    ;ahora envia la amplitud
          OUT  (&HA1),A
          LD   B,4                    ;asigna un bucle de retardo
RETARDO1  LD   HL,&HFFFF
RETARDO2  DEC  HL
          LD   A,H
          OR   L
          JR   NZ,RETARDO2
          DJNZ RETARDO1               ;en el tiempo de retardo
                                      ;se oye el tono
          LD   A,&H08
          OUT  (&HA0),A               ;envia  la direccion  del re-
DESCONEC  LD   A,0                    ;gistro de amplitud y enton-
          OUT  (&HA1),A               ;ces fija la amplitud a cero
          EI
          RET
```

Si lees con atención los comentarios que hay en la rutina, serás capaz de entender lo que hace ésta una vez que encendemos el sonido, éste continuará hasta que lo apaguemos. Asignando un 0 al registro de amplitud del canal apropiado, intenta modificar el programa anterior sustituyendo a partir de las etiquetas DESCONEC por la rutina siguiente:

```
DESCONEC  LD    C,8
          LD    B,8
BUCLE2    LD    HL,&HFFFF
BUCLE1    DEC   HL
          LD    A,H
          OR    L
          JR    NZ,BUCLE1
          DEC   C
          LD    A,&H08 ; selecciona el registro para escribir
          OUT   (&HA0),A ; 8
          LD    A,C
          OUT   (&HA1),A ; manda el dato
          DJNZ  BUCLE2
          EI
          RET
```

¿Entiendes lo que hace esta rutina? En vez de desconectar de repente el canal 1, hace que el sonido se vaya apagando, disminuyendo gradualmente el valor enviado al registro de amplitud del canal 1.

El registro C se inicializa con el valor 8 y se va disminuyendo de uno en uno cada vez que se ejecuta el bucle. Después, el valor resultante se escribe en el registro 8 del PSG. Si te preguntas por qué he inicializado el registro C cuando el registro B ha sido decrementado, la respuesta es muy simple: cuando el registro B tenga un 0 en la instrucción DJNZ, el Z-80 ejecutará la instrucción EI, dejando un 1 en el registro 8, y el tono que saldrá será todavía audible.

También puedes cambiar el contenido de los registros de control de tono mientras se esté produciendo un sonido, con lo cual se alteraría la frecuencia tan pronto como los nuevos valores del grado se escriban en los registros del PSG. Por ejemplo, sustituye la rutina a partir de la etiqueta DESCONEC anterior por la rutina siguiente, que cambia lentamente el valor del registro de control del tono.

```
DESCONEC  LD    C,8
          LD    B,8
BUCLE1    LD    HL,&HFFFF
BUCLE2    DEC   HL
          LD    A,H
          OR    L
          JR    NZ,BUCLE2
          LD    A,&H01 ; selecciona control de graves
          OUT   (&HA0),A ; el grado del tono se manda al PSG
          LD    A,C
          OUT   (&HA1),A ; manda el grado
          DJNZ  BUCLE1
          LD    A,8 ; amplitud del canal 1
          OUT   (&HA0),A ; número de registro
          LD    A,0 ; apaga el canal 1
          OUT   (&HA1),A
          EI
          RET
```

Esta nos dará una serie de tonos diferentes. Si cambias el valor del registro de control fino de tono, obtendrás una especie de intermitencia de sonidos. Prueba ahora a escribir una rutina en código máquina que cambie desde 0 hasta su máximo valor el registro de control de tono de 12 bits, cuyo resultado será una disminución progresiva de la frecuencia del sonido.

Hasta ahora todo el trabajo que hemos estado realizando ha sido con el canal 1 del PSG; la razón es porque no hay diferencias entre los tres canales, por tanto, puedes aplicar a los canales 2 y 3 lo que has aprendido con el canal 1. Además, puedes conseguir sonidos a la vez por los tres canales simplemente asignando por turnos los registros para cada canal. En código máquina no existe realmente pausa entre el sonido que emite un canal y los otros.

Emitir un sonido por el canal 1 y otro por el canal 2 implica los siguientes pasos:

1. Escribir la información del tono para el canal 1 en los registros 0 y 1.
2. Escribir la información del tono para el canal 2 en los registros 2 y 3.
3. Escribir la información sobre la amplitud para el canal en el registro 8.
4. Escribir la amplitud para el canal 2 en el registro 9.

Los dos tonos empezarán al mismo tiempo.

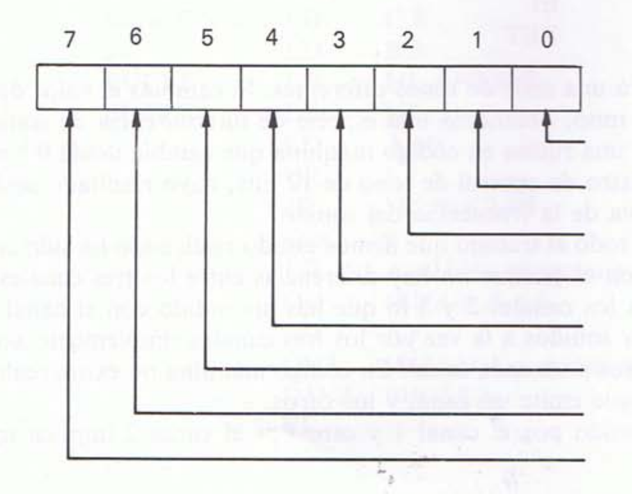
Cómo conectar y desconectar canales

Ya hemos visto antes cómo apagar el sonido en los canales del PSG simplemente fijando la amplitud del correspondiente registro a 0. El único problema que acarrea esta técnica es si quieres conectar el canal con la misma amplitud o si quieres emitir ruidos por ese canal y no una nota. Con el registro 7 puedes inhibir la salida de tono de un canal; dicho registro es el principal registro de control del PSG. Si inhibes la emisión de un tono o de un ruido en un canal determinado, se dice que estás desactivando el tono o el ruido de dicho canal. Si, por el contrario, permites la emisión de tonos o de ruidos por un canal, se dice que estás activando el ruido o el tono en ese canal.

El registro 7

Aunque también otros registros tienen información sobre el tono y la amplitud del sonido que van a emitir los diferentes canales del PSG, el registro 7 actúa como un registro de control.

Es un registro de 8 bits; cada bit controla una parte del funcionamiento del PSG. En el siguiente diagrama se ve muy bien la función de cada bit.



El bit 0

Es el bit de activación del tono del canal 1. Cuando este bit se activa, la salida del tono del canal 1 se inhibe, con lo que no se emitirá sonido, es decir, silencio, sin importar el valor del registro 8.

Si asignas a este bit un 0, el resultado será un tono que emita el canal 1 con el volumen especificado en el registro de control de amplitud de canal 1, registro 8. Obviamente, si se fija a 0 el registro 8, no saldrá ningún sonido del canal 1, aunque este bit esté a 0 o a 1.

El bit 1

Este bit hace la misma función para el canal 2 que el bit 0 para el canal 1.

El bit 2

Este bit hace lo mismo que el bit 0, pero para el canal 3.

El bit 3

Sobre el tema de los ruidos hablaré un poco más adelante. Cuando a este bit se le asigna un 0, el canal 1 emite un ruido con la amplitud que indique el registro 8. Cuando este bit se active, no se oirá ningún ruido. Es posible tener activados ruido y tono a la vez en cualquiera de los canales; puedes hacerlo, en el canal 1, simplemente asignando un 0 a los bits 0 y 3 del registro 7. Si activas estos dos bits a 1, se inhibirá cualquier sonido del canal 1.

El bit 4

Es el bit de activación de ruido del canal 2.

El bit 5

Es el bit de activación de ruido del canal 3.

Los bits 6 y 7

Ya dije antes que el generador de sonido programable tiene dos registros que los usaremos como puertas de entrada/salida, de forma análoga a los registros

del PPI. A los dos registros se les suele denominar, usando toda la imaginación que sólo los diseñadores electrónicos parecen tener, A y B.

El A es el registro 14 del PSG y el B es el registro 15. Al igual que los registros dentro del PPI, estas puertas pueden ser tanto de entrada como de salida, dependiendo del estado de los siguientes 2 bits del registro 7.

Bit 6	*Bit 7*	*Puerta A*	*Puerta B*
0	0	ENTRADA	ENTRADA
0	1	ENTRADA	SALIDA
1	0	SALIDA	ENTRADA
1	1	SALIDA	SALIDA

El sistema operativo del MSX utiliza estas dos puertas para que vigilen los mandos de juegos, pero realmente no vas a necesitar trabajar con estos registros.

Debes dejar el bit 6 a 0 y el bit 7 a 1. El registro de entrada puede leerse como antes señalé, accediendo a los otros registros del PSG; sin embargo, el contenido del registro y su correspondencia con el *joystick* se salen de los propósitos de este libro.

Cómo acceder al registro 7

Debido a que el registro 7 es un conjunto de bits que controlan varios aspectos del funcionamiento del PSG, es importante que sólo modifiques los bits del registro que realmente quieras cambiar. Esta operación la puedes realizar leyendo el contenido del registro en el registro A de la UCP, modificar los bits y después escribir el nuevo valor otra vez en el registro.

Puedes usar las instrucciones AND y OR para activar y desactivar los bits o si lo prefieres las instrucciones SET y RESET del Z-80. El siguiente listado es un ejemplo de cómo emitir tono y ruidos a la vez por el canal 1 del PSG.

```
LD      A,7
OUT     (&HA0),A
IN      A,(&HA2)
RES     0,A
RES     3,A
LD      C,A
LD      A,7
OUT     (&HA0),A
LD      A,C
OUT     (&HA1),A
```

Los ruidos

Anteriormente ya he mencionado la palabra ruido blanco; este tipo de ruidos es como un silbido o un siseo agudo similar, por ejemplo, al del canal de onda media cuando no recibe ninguna emisora. Muchos de los sonidos creados por el hombre e incluso por la naturaleza tienen un gran porcentaje de ruido; por ejemplo: el viento, la lluvia, silbatos de vapor, etc. Los tres canales que ya hemos visto son capaces de emitir ruidos, ya sea a la vez o en lugar de un sonido. La amplitud de dichos ruidos está controlada por el registro de control de amplitud para cada canal. Antes de que puedas oír un ruido en un canal es necesario activar el ruido poniendo a 0 el bit del registro 7 adecuado. El siguiente ejemplo consigue que el canal 1 emita ruido blanco.

EJEMPLO 42

Cuando ejecutes esta rutina, el canal 1 emitirá un ruido hasta que se pongan a 0 los registros correspondientes del PSG. Pulsando CTRL-G, se consigue esto.

```
LD   A,7                                 ;coge el contenido del regis-
OUT  (&HA0),A                            ;tro 7  en ese  momento y lo
IN   A,(&HA2)                            ;modifica
SET  0,A                                 ;desactivacion del tono
RES  0,A                                 ;activacion del ruido
LD   C,A
LD   A,7
OUT  (&HA0),A
LD   A,C
OUT  (&HA1),A                            ;manda el nuevo registro 7
LD   A,8
OUT  (&HA0),A                            ;envia la amplitud del canal1
OUT  (&HA1),A
RET
```

Al igual que los sonidos, los ruidos también tienen tonos. El tono de un ruido se define como la frecuencia, alta o baja, presente en el ruido. Si la frecuencia es alta, oirás un siseo más bien suave, mientras que, si la frecuencia del sonido es baja, el ruido será como un silbido fuerte. El contenido del registro 6 es el que determina el grado del ruido que va emitir el PSG; este registro es el de control de tono del ruido.

El registro 6

El hecho de que sólo haya un registro para controlar el tono de un ruido indica que todos los canales emitirán un ruido con la misma frecuencia.

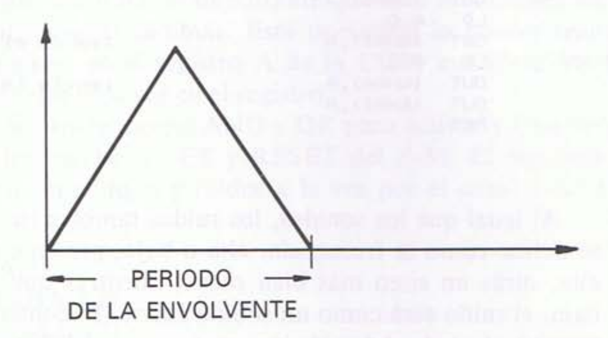
Este registro tiene 5 bits; por tanto, los valores del tono van desde 0 a 31. El 31 es el grado más bajo de ruido, mientras que el 0 es el más alto. Si modificas el valor de este registro, también cambiará la señal del sonido que se va a oír.

Envolventes

¿Cuál es la diferencia entre una misma nota emitida, por ejemplo, por un piano o por una flauta? Pues bien, aunque el tono sea el mismo, la nota sonará diferente. Al igual que por su tono, el sonido de una nota está determinado también por su amplitud y por cómo estas dos características cambian su valor mientras se está emitiendo la nota. Por ejemplo, en el gráfico siguiente se ve muy bien cómo la amplitud de un tono varía en el tiempo. El tono del gráfico es del tipo al que hemos visto hasta ahora:



El volumen del tono va desde 0 hasta su valor máximo, donde permanece constante hasta que vuelve otra vez al 0. Si pudiéramos dibujar la amplitud de una nota, como en la figura siguiente, oiríamos un sonido bastante extraño.



En este gráfico vemos un incremento gradual de la amplitud seguido por una disminución. A este tipo de sonido se llama ENVOLVENTE. En los ordenadores MSX hay ocho ENVOLVENTES diferentes. En otros ordenadores con generadores de sonido programables más sofisticados es posible construir las envolventes para lograr un sonido determinado.

Lo más importante que debes recordar acerca de estas envolventes es que, una vez que se le indica al PSG que utilice una envolvente escribiendo el correspondiente valor en los registros de control de envolventes, el sonido se generará bajo el control de la envolvente y no bajo el del usuario. El tipo de envolvente aplicada al sonido que se está generando depende del contenido del registro 13.

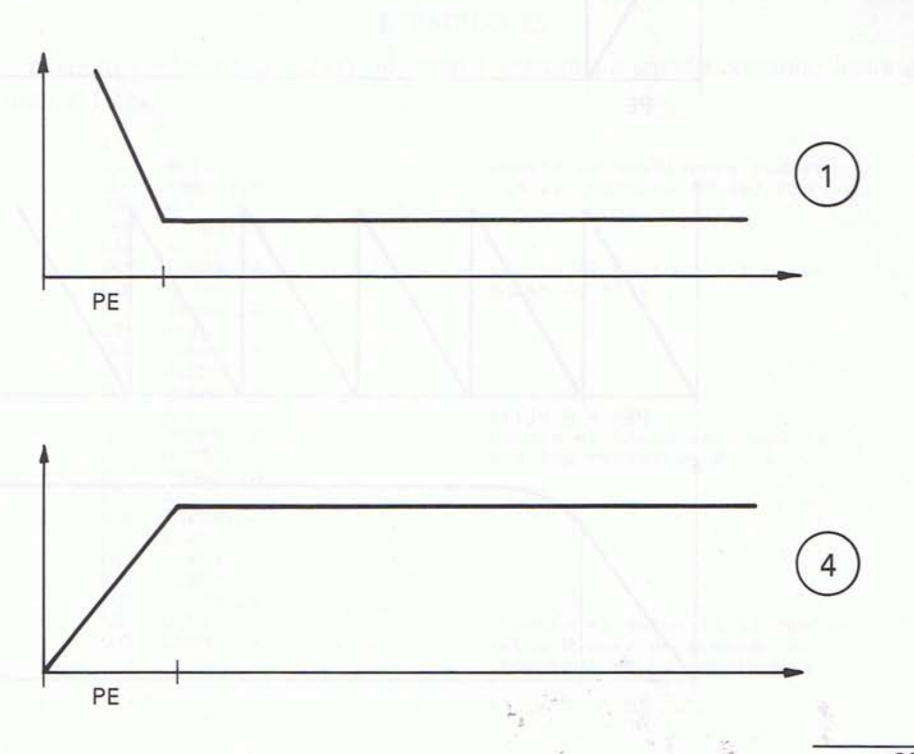
El registro 13

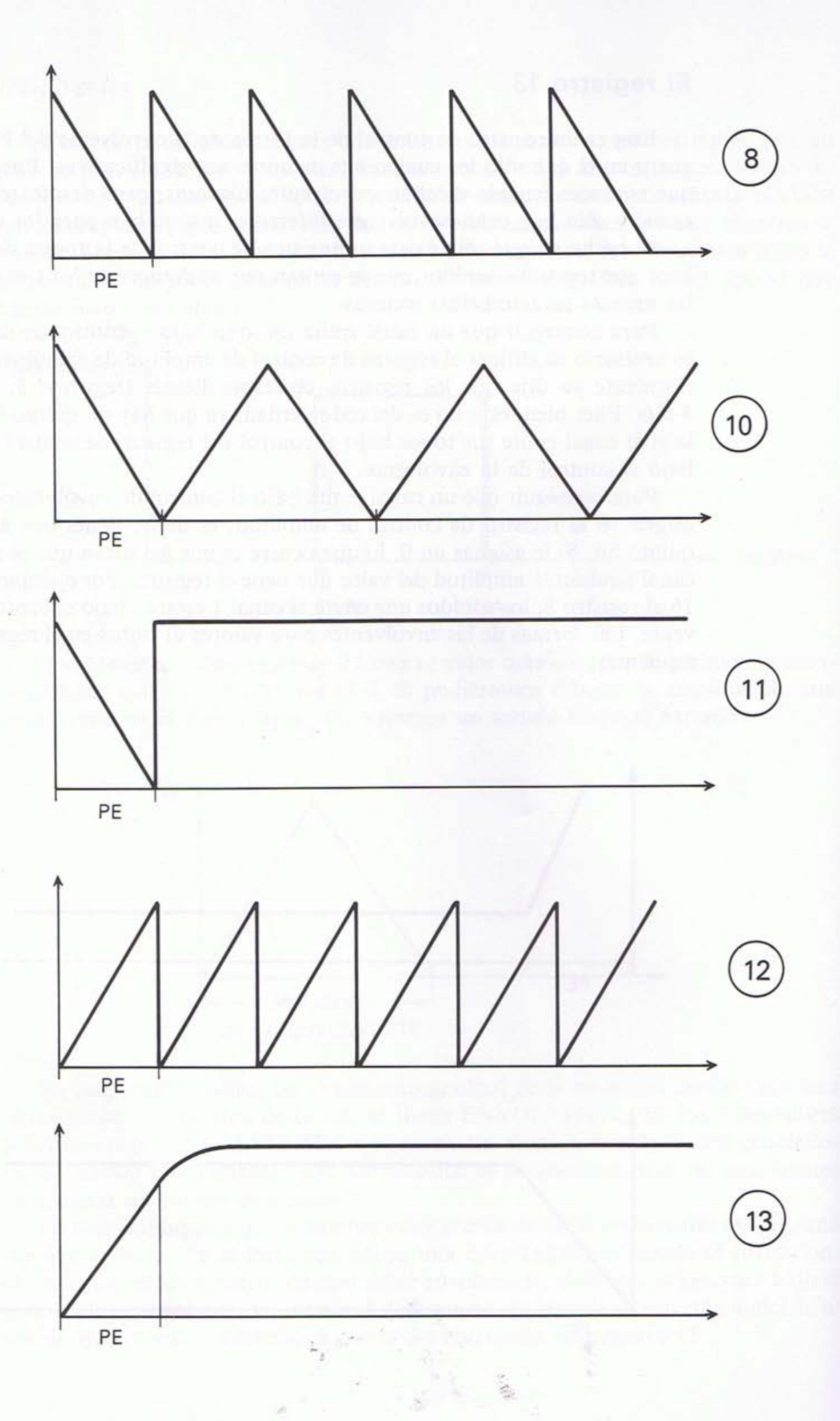
Este es un registro de control de la forma de la envolvente del PSG y es un registro en el que sólo los cuatro bits menores son significativos. Puede que pienses que entonces tendrás dieciséis envolventes distintas, pero desafortunadamente no es así y sólo hay ocho envolventes diferentes disponibles para los usuarios.

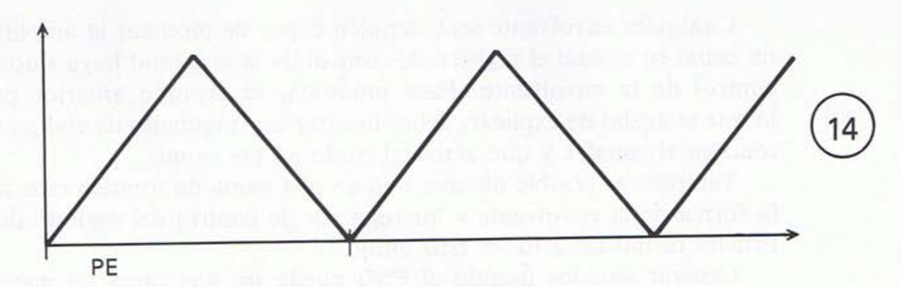
El hecho de que sólo exista un registro de control de la forma de la envolvente hace que todos los sonidos que se emitan por cualquiera de los tres canales tengan las mismas características sonoras.

Para conseguir que un canal emita un tono bajo el control de una envolvente, es necesario modificar el registro de control de amplitud de ese mismo canal. Anteriormente ya dije que los registros correspondientes (registros 8, 9, 10) son de 4 bits. Pues bien, esto no es del todo verdad, ya que hay un quinto bit que controla si el canal emite sus tonos bajo el control del registro de control de amplitud o bajo el control de la envolvente.

Para conseguir que un canal emita bajo el control de envolventes, simplemente asigna 16 al registro de control de amplitud, es decir, tienes que asignar un 1 al quinto bit. Si le asignas un 0, lo que ocurre es que los tonos que se emitan por ese canal tendrán la amplitud del valor que tiene el registro. Por ejemplo, si asignas un 16 al registro 8, los sonidos que emite el canal 1 estarán bajo el control de la envolvente. Las formas de las envolventes para valores distintos en el registro 13 son las siguientes:




8
PE
10
PE
11
PE
12
PE
13
PE



En todos estos ejemplos, PE es la abreviatura de Período de la Envolvente, que es una unidad de medida que indica el tiempo que tarda en ejecutarse un ciclo de la envolvente. Cambiando los valores de los registros 11 y 12, puedes variar este parámetro y, por tanto, variar la proporción del cambio de la amplitud bajo el control de la envolvente.

Los registros 11 y 12

Los dos juntos forman el registro de CONTROL DEL PERIODO DE LA ENVOLVENTE de 16 bits. El registro 11 tiene los 8 bits menores y el registro 12 los 8 bits mayores. Cuanto más grande sea el valor de este registro de 16 bits, más largo será el período de la envolvente. Con el siguiente ejemplo aprenderás cómo programar el PSG para que emita sonidos por el canal 1 bajo el control de la envolvente 14.

EJEMPLO 43

Cuando ejecutes esta rutina, el canal 1 emitirá un sonido continuo hasta que pulses CTRL-G.

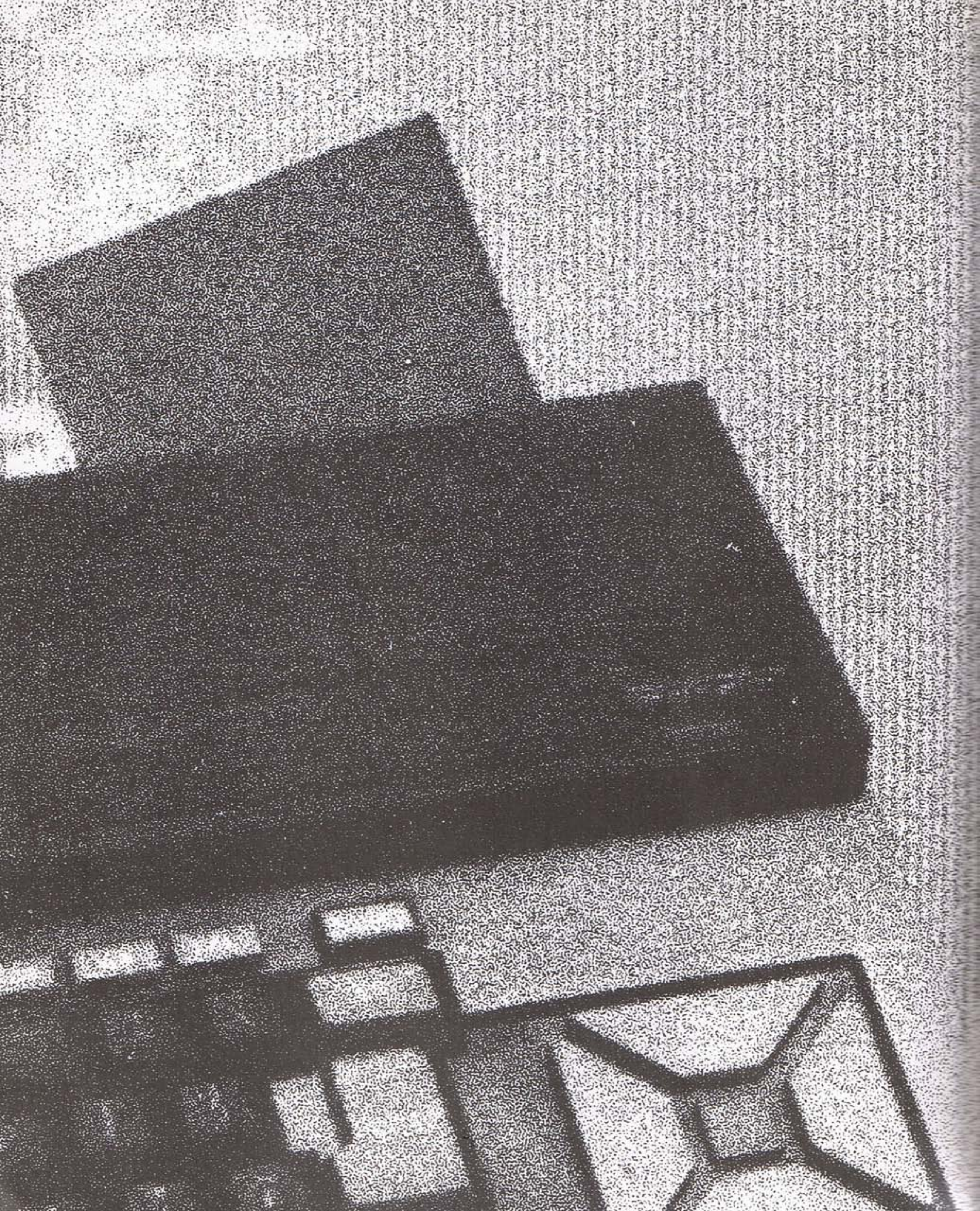
```
DI
LD   A,13                     ;envia la envolvente numero
OUT  (&HA0),A                 ;14 al registro 13 del PSG
INC  A
OUT  (&HA1),A
LD   A,11
OUT  (&HA0),A                 ;manda el periodo a los re-
LD   A,255                    ;gistros 11 y 12
OUT  (&HA1),A
LD   A,12
OUT  (&HA0),A
LD   A,255
OUT  (&HA1),A
INC  A                        ;fija A a 0
OUT  (&HA0),A                 ;envia el grado del canal 1
LD   A,255                    ; a los registros 0 y 1
OUT  (&HA1),A
LD   A,1
OUT  (&HA0),A
LD   A,8
OUT  (&HA1),A
LD   A,8
OUT  (&HA0),A
LD   A,16                     ;envia el valor 16 al regis-
OUT  (&HA1),A                 ;tro 8 para determinar el
                              ;control de la envolvente
EI
RET
```

Cualquier envolvente será también capaz de modular la amplitud del ruido de un canal en el cual el registro de control de la amplitud haya sido asignado por el control de la envolvente. Para modificar el ejemplo anterior para que realice lo que te acabo de explicar, debes insertar un fragmento de código que desactive el tono en el canal 1 y que active el ruido en ese canal.

También es posible obtener una amplia gama de sonidos especiales cambiando la forma de la envolvente y los registros de control del período de la envolvente. Prueba tú mismo a hacer esto último.

Generar sonidos usando el PSG puede ser una tarea un poco frustrante, ya que, para conseguir los efectos deseados, necesitarás experiencia y tiempo.

Con esto doy por terminado el capítulo sobre las posibilidades para emitir sonido que te ofrece tu ordenador MSX. Espero que te sientas lo suficientemente seguro de tus habilidades como programador y empieces ya mismo a programar sonidos con este código máquina. En el último capítulo del libro he incluido algunas ideas y notas sobre este código máquina, pues realmente no es necesario dedicarles un capítulo entero.



12

Temas diversos

Como ya te he dicho antes, en este capítulo he reunido algunos conceptos e ideas sobre el MSX que no necesitan un capítulo entero cada uno de ellos. En realidad no es absolutamente necesario que leas este tema para ser capaz de empezar a programar en este lenguaje máquina, pero espero que lo estudies a fondo más adelante si vas a seguir más de cerca el mundo de la programación.

Enlaces

Normalmente la CPU Z-80 está siempre ejecutando el intérprete MSX BASIC o el sistema operativo, ambos situados en la ROM de tu ordenador. La UCP Z-80, como ya sabes, deja de operar con esos dos programas cuando ejecutas una instrucción USR de una de tus rutinas de la RAM, pero esto también ocurre durante la ejecución de determinadas rutinas de la ROM que llaman a direcciones de la RAM. Estas rutinas están formadas por algunas de las rutinas más importantes del sistema operativo y se llaman HOOK CALL, que en español significa LLAMADAS DE ENLACE.

En condiciones normales, la posición de la RAM, situada en la dirección llamada por una rutina ROM, tiene en ella el código de la instrucción RET. Normalmente, por tanto, la instrucción CALL ejecuta la RET que pasaría inmediatamente el control a esa rutina de la ROM. Si cambias la instrucción RET por una

instrucción CALL o JP, conseguirás que el sistema operativo haga operaciones totalmente diferentes antes de continuar con sus "obligaciones" normales.

Zona de servicio de enlaces

A la dirección a la que llama una instrucción HOOK CALL de la ROM y a los 4 bytes siguientes a ella se les llama ZONA DE SERVICIO DE ENLACES para esa instrucción determinada. Por ejemplo, la zona de memoria de la RAM de 5 bytes a partir de la dirección &HFD9A es la zona de servicio de enlaces para la llamada de enlace dentro de la rutina que la interrupción VDP llamó. Obviamente, esta zona se llama cada vez que el VDP genera una interrupción, teniendo en cuenta que estén activadas las interrupciones en la UCP. De esta manera, si sustituyes la instrucción RET por la CALL, te habrás diseñado una subrutina que se ejecutará cada vez que ocurra una interrupción en el VDP.

Por ejemplo:

```
&HFD9A    CALL    59000
          RET
          RET
```

ejecutará la rutina que empieza en la dirección 59000. Las dos instrucciones RET ya están en la zona de servicio. La rutina a la que se llama tiene que terminar, como ya supones, con una instrucción RET para devolver el control a la ROM. Normalmente es aconsejable guardar todos los registros de la UCP cuando utilices rutinas de servicio de enlaces. En este caso particular la ROM guarda todos los registros antes de ejecutar la instrucción HOOK CALL y los devuleve cuando vuelvas al programa de la ROM.

En este caso particular, y debido a que la rutina de servicio de enlaces se ejecuta cada dos centésimas de segundo, es importante que ésta no sea muy larga. Y ya por último escribe la rutina de servicio de enlace en la dirección apropiada antes de modificar la zona de servicio de enlace correspondiente, ya que, de lo contrario, podrías perder el control del ordenador.

Zona de trabajo del sistema

Seguro que ya te has dado cuenta de que existen amplias zonas de memoria que sólo usa el ordenador, en este caso los MSX. Esta área, a salvo de los estragos del BASIC, se denomina ZONA DE TRABAJO DEL SISTEMA y el ordenador la utiliza como papel "en sucio" para las rutinas de la ROM; por ejemplo, las copias de la RAM del contenido de los registros VDP se guardan aquí junto con las definiciones de las teclas y algunos que otros bits. Debido a su importancia para el programa de la ROM, esta zona sólo debes modificarla si estás muy seguro de lo

que vas a hacer. Desde luego NUNCA debes utilizarla para almacenar programas en código máquina.

En la RAM de los ordenadores estándares MSX, la zona de trabajo del sistema empieza en la posición &HF380, por lo que debes tener cuidado con cualquier posición por encima de esa.

Estructura de las variables del BASIC

Ya vimos al principio del libro cómo podías devolver resultados de un programa en código máquina a las variables del BASIC usando la rutina USR. Sin embargo, sería bastante útil poder acceder a las variables de tus rutinas sin tener que usar la zona de parámetro; por ejemplo, suponte que quieres mandar más de un parámetro a un programa en código máquina; lo que se hace en esos casos es asignar, con la instrucción POKE, el segundo parámetro en la RAM para que pueda accederse a él más adelante; de todas formas, es mucho más elegante en la mayoría de los casos utilizar una variable en BASIC. En este último capítulo te indico la estructura de los tipos de variables enteras y alfanuméricas. Las variables reales no las voy a explicar.

Puedes encontrar la dirección de cualquier variable que quieras usando la instrucción VARPTR del BASIC. Date cuenta de que esta instrucción tendrá la dirección de la longitud de la cadena cuando la uses para encontrar la dirección de una variable alfanumérica, y tendrá el valor de la variable cuando la utilices para variables enteras.

Variables alfanuméricas

3	identificador del tipo de variable
n$	primer carácter del nombre
n1$	segundo carácter del nombre
L	longitud de la cadena
low	byte bajo de la dirección de la cadena
high	byte alto de la dirección de la cadena

Seguro que ya te has dado cuenta de la similitud que existe entre los tres últimos bytes de la variable y la zona de descripción de la cadena que usamos para pasar parámetros alfanuméricos a rutinas en código máquina.

Matrices alfanuméricas

3	identificador
n$	primer carácter del nombre
n1$	segundo carácter del nombre
?	

?	
DIM	número de dimensión de la matriz
elo	byte bajo del número total de elementos
ehi	byte alto del número total de elementos
len0	longitud del elemento 0
elo0	byte bajo de la dirección del elemento 0
ehi0	byte alto de la dirección del elemento 0

Las últimas tres entradas en la tabla se repiten en cada elemento de la matriz. Las direcciones del contenido de los elementos, especificado en *elo0* y en *ehi0,* están a menudo, si no siempre, en la zona de la cadena que haya sido dimensionada. Si tanto la longitud como la dirección de las entradas de un determinado elemento de la matriz están a 0, esto quiere decir que todavía no se les ha asignado todavía un valor.

Variables enteras

Este tipo de variables están dispuestas en la memoria de forma muy sencilla:

2	identificador
n$	primer carácter del nombre
n1$	segundo carácter del nombre
val low	byte bajo del valor
val high	byte alto del valor

Recuerda que las variables enteras del MSX tienen signo, lo cual significa que cualquier valor mayor a 32767 será un número negativo representado en complemento a dos.

Matrices enteras

En líneas generales, la estructura de este tipo de matrices es similar a la de las matrices alfanuméricas que te he explicado un poco antes:

2	identificador
n$	primer carácter del nombre
n1$	segundo carácter del nombre
?	
?	
DIMS	número que indica la dimensión de la matriz
elo	byte bajo del número total de elementos
ehi	byte alto del número de elementos
ele0low	byte bajo del valor del primer elemento
ele0high	bite alto del valor del primer elemento

Las dos últimas entradas se repiten en cada elemento de la matriz.

Apéndice 1

Cómo introducir código máquina

El objetivo de este apéndice es el de enseñarte algunos métodos para introducir en tu ordenador MSX programas en código máquina. El primer método que vamos a ver consiste en introducir un sencillo monitor de código máquina. Un monitor es un programa que ofrece al usuario la posibilidad de ver lo que hay en las posiciones de memoria. Además, puedes modificar dichas posiciones, mover bloques de bytes dentro de la memoria y escribir y leer zonas de memoria de un cassette.

Primero te he listado el programa que he usado para introducir las rutinas de este libro.

```
10 REM PROGRAMA MONITOR
20 REM Joe Pritchard
30 REM Septiembre 1985
50 CLEAR 200,57343!:KEYOFF
60 GOSUB 1000
70 GOSUB 1100
80 ON I GOSUB 2000,5000,4000,3000,6000,7000
90 GOTO 70
1000 SCREEN 0
1005 START=57344!
1010 RETURN
1100 CLS:WIDTH 28:LOCATE 10,0,0:PRINT "Opciones"
1115 LOCATE 0,6,0
1120 PRINT "1.-Entrada Codigo Maquina"
1130 PRINT "2.-Almacenar Codigo Maquina"
1140 PRINT "3.-Cargar Codigo Maquina"
1150 PRINT "4.-Listado Codigo Maquina"
1160 PRINT "5.-Traslado del Codigo"
```

```
1161 PRINT "6.-Ejecutar Codigo"
1170 PRINT:PRINT:PRINT:PRINT "  Digite una opcion: ";
1180 I$=INPUT$(1) :I=ASC(I$):IF I<49 OR I>54 THEN GOTO 1180
     ELSE I=I-48
1190 RETURN
2000 CLS:INPUT "escribir en : ";D$
2015 GOSUB 10000
2020 IF FI<START OR FI>61000! THEN BEEP:BEEP:GOTO 2000
2030 PRINT "Digite los numeros en Hexad."
2040 GOSUB 8000
2045 IF DA=999 THEN RETURN
2050 PRINT HEX$(FI);"     ";HEX$(DA)
2055 POKE FI,DA
2060 FI=FI+1:GOTO 2040
3000 CLS:INPUT "Direccion Listado: ";D$:GOSUB 10000
3010 FOR I=FI TO FI+10:PRINT HEX$(I);"     ";HEX$(PEEK(I)):NEXT
3015 PRINT :PRINT "Digite X para salir":X$=INPUT$(1)
3020 IF X$="x" OR X$="X" THEN RETURN ELSE FI=I:GOTO 3010
4000 CLS :INPUT"Nombre del Fichero: ";N$:PRINT "Buscando el
     Fichero: "+N$
4020 BLOAD N$
4030 RETURN
5000 CLS:INPUT"Almacenar desde : ";D$:GOSUB 10000:A=FI
5010 INPUT"Numeros de bytes : ";D$:GOSUB 10000:N=FI
5020 INPUT "Nombre del fichero ";N$
5025 IF N$="" THEN GOTO 5020
5030 BSAVE N$,A,A+N+1
5040 RETURN
6000 CLS:INPUT"Trasladar desde : ";D$:GOSUB 10000:F=FI
6010 INPUT"Hasta direccion:";D$:GOSUB 10000:U=FI
6020 INPUT"Nueva direccion:";D$:GOSUB 10000:T=FI
6030 IF T>F THEN GOTO 6500
6040 MOVE=T
6050 FOR I=F TO V:POKE MOVE,PEEK(I):MOVE=MOVE-1:NEX:RETURN
6500 MOVE=U+T-F
6510 FOR I=U TO F STEP-1:POKE MOVE,PEEK(I):MOVE=MOVE-1:NEXT:RETURN
7000 CLS:INPUT "Ejecutar desde: ";D$:GOSUB 10000:A=FI
7010 CLS:DEFUSR=A:L=USR(0)
7020 RETURN
8000 A$=INPUT$(2):IF A$="qq" OR A$="QQ" THEN DA=999:RETURN ELSE
     A$="&HO"+A$:DA=V
AL(A$):RETURN
10000 IF VAL(D$)<0 THEN FI=VAL(D$)+65536! ELSE FI=VAL(D$)
10110 RETURN
```

Introduce este programa y guárdalo en un cassette. Para ejecutarlo, simplemente teclea la palabra RUN y pulsa RETURN.

Cómo usar el programa

Cómo introducir código máquina

Esto es lo primero que normalmente querrás hacer con el programa anterior. Introduce la opción 1 del menú. Cuando te salga la pregunta

ESCRIBIR EN:

podrás introducir la dirección, que va a ser la primera posición en la RAM. Dicha dirección tendrá el código máquina. Por ejemplo, donde en el libro dije que escri-

bieras el código en la dirección 59000, debes teclear el número 59000 en respuesta a la pregunta ESCRIBIR EN: y después pulsa RETURN. Los números en hexadecimal se introducen escribiendo "&H" antes del número; por ejemplo, si escribes &HE290, la primera dirección se escribirá en la posición 58000.

Supongo que ya estás preparado para introducir los bytes de cualquier programa en código máquina. Fíjate que los bytes que introduzcas TIENEN que estar en hexadecimal. El programa acepta dos pulsaciones de teclas y después, con la instrucción POKE, asigna el valor resultante en la posición de memoria direccionada en ese momento; luego se incrementará el contador de posiciones de memoria. Por ejemplo, si introdujeras el valor 1 en el ordenador, deberías introducir "01", en vez del valor decimal 32 tendrías que introducie el "20", y en vez del valor 201, tendrías que introducir el "C9", etc.

Fíjate que en este caso no es necesario el prefijo "&H", al igual que tampoco lo es pulsar la tecla RETURN. Una vez que hayas introducido todos los hexadecimales, pulsa la tecla "Q" dos veces para volver al menú.

En caso de que te hayas confundido al introducir el código, simplemente vuelve a introducir la opción "Entrada código máquina" con la dirección en que cometiste el error. Por ejemplo, supónte que quieres introducir "C8" y en su lugar introdujeses la letra "V" como primera pulsación; lo que tendrías que hacer entonces es pulsar la tecla "8" para terminar esa entrada, con lo que la dirección aparecería junto con el valor "08". Apunta la dirección y deja esa opción pulsando dos veces la tecla "Q". Vuelve a introducir la opción y entonces especifica la dirección en la que vas a escribir el código, es decir, la dirección en la que cometiste la equivocación; después pulsa "C" y "8", que son el valor correcto, y luego introduce normalmente el resto de programa.

Puede que esto te parezca un poco complicado, pero te permite rapidez y precisión en la entrada de datos, como es lógico, teniendo en cuenta que debes introducir todos los números como números de dos dígitos, añadiendo un 0 al principio del número en caso de que el número que quieras introducir sólo tuviera un dígito. Practica esta opción y seguro que pronto te acostumbrarás a ella.

Cómo guardar código máquina

Cuando hayas escrito un fragmento de código máquina, esta opción te permite guardarlo en una cinta. En cuanto los datos estén grabados en la cinta, volverás al menú.

Cómo cargar código máquina

Esta rutina permite recobrar de la cinta fragmentos de código máquina. Escribe el nombre del fichero como respuesta a la pregunta que te aparecerá en la pantalla, conecta después la cinta y ya puedes comenzar a cargar. Una vez que hayas cargado el fichero en el ordenador, el programa volverá al menú. El código se cargará en la dirección en la que fue guardado en la cinta.

Cómo listar código máquina

Simplemente introduce la dirección de la primera posición, en hexadecimal o en decimal de la que quieras ver el contenido. Después el programa listará, en hexadecimal, el contenido de la dirección y de las diez direcciones siguientes. Si quieres terminar el listado, pulsa la tecla "X" en cualquier momento. Para ver la siguiente zona de diez posiciones, pulsa la barra espaciadora.

Cómo mover código máquina

Esta opción nos permite mover bloques de bytes en la memoria. Este truco es muy útil para, por ejemplo, insertar bytes en un determinado programa sin tener que escribir encima del programa.

Veamos un ejemplo típico de esto:

58000	1	1
	2	2
	3	3
	4	20
	5	4
	6	5
	SIN USAR	6
	ORIGINAL	RESULTADO DESEADO

En este ejemplo vemos que lo que necesito es mover un byte hacia abajo los bytes a partir de la dirección 58003 hasta la dirección 58005 para así poder insertar el valor "20". Para realizar este movimiento de bytes, introduce el valor 58003 como respuesta a la pregunta "Trasladar desde:", el valor 58004, como respuesta a la pregunta "Nueva dirección".

La pregunta "Nueva dirección" se refiere, como ves, a la posición a la que vas a mover el primer byte del bloque.

Ensaya con este programa hasta que hayas asimilado las diferentes facilidades que te ofrece. Para ejecutar un programa en código máquina que haya sido introducido con este programa puedes pararlo pulsando CRTL-STOP; ejecuta la instrucción DEFUSR para asignar la dirección de comienzo del código que va a ejecutarse y después ejecuta el código con una instrucción USR.

Cómo incluir código máquina en un programa

A veces resulta muy útil incorporar en los programas en BASIC subrutinas en código máquina. Hay dos formas que el programador puede utilizar:

La primera de ellas consiste en guardar en una cinta los bytes del código máquina y volver a cargar los bytes en el ordenador mientras se está ejecutando el programa. La segunda forma consiste en poner los bytes que forman el programa

en el de BASIC como una, o más, instrucción de datos. Puedes usar un bucle FOR ... NEXT para leer los bytes de las instrucciones de datos y asignarlos con la instrucción POKE en las posiciones deseadas de la RAM.

Cómo escribir programas en código máquina

Los principios fundamentales para realizar buenos programas en código máquina no son muy diferentes a los que aplicas para escribir buenos programas en BASIC. Seguidamente paso a explicarte algunos aspectos importantes que debes tener en cuenta a la hora de escribir rutinas en código máquina; no importa lo largas que éstas sean para que funcionen lo mejor posible.

Especificaciones

Cualquier programa en código máquina que vayas a escribir debes comenzarlo habiendo especificado por escrito antes lo que quieres que el programa haga para que así sepas exactamente lo que tienes que hacer. Al principio te será bastante difícil, si no imposible, escribir un programa en código máquina directamente con el teclado, por lo que es normal que todos los programas en código máquina empiecen en una hoja de papel. Además, teniendo por escrito lo que se llama el "cuerpo principal" del programa, evitarás cargar el programa de bits inútiles mientras lo vayas desarrollando. Por esto, conviene que tus programas sean concretos y claros, con lo cual te evitas dificultades a la hora, por ejemplo, de tener que volver a repasar tus rutinas o de que otros las entiendan.

Codificación

Una vez que ya hayas creado este cuerpo principal, ya puedes empezar a dividir el programa en pequeños fragmentos para luego ir escribiendo el código máquina necesario que ejecute la labor que hayas escrito. Si alguna de esas partes te ocasiona algún problema, lo mejor es considerarlo como si fuera un programa entero y volver a dividirlo en pequeñas partes nuevamente.

De esta manera cada parte es programada y probada antes de incorporarla al programa principal. Lo más importante que debes recordar cuando ensambles código a mano son estas tres cosas que ahora paso a explicarte:

1. Repasa todo el trabajo con atención: cuando ensambles a mano un programa es muy corriente que te equivoques al leer los números que tienes en las tablas del final del libro.
2. Comprueba con atención la dirección de todas las instrucciones JP y CALL y asegúrate de que son correctos los bytes de desplazamiento en los saltos relativos.
3. Por último, guarda siempre en cintas el código antes de ejecutarlo para que puedas volver a cargarlo en caso de que algo esté mal, que seguro que te ocurrirá multitud de veces.

Apéndice 2

Rutinas de tiempo

En muchas de las tablas de instrucciones Z-80 a lo largo de este libro te he dado los tiempos que tardan en ejecutarse las instrucciones del Z-80. Estos tiempos se miden en "ciclos" del reloj que dirige la UCP. En los ordenadores MSX se generan 3,57 millones de pulsaciones del reloj cada segundo, lo cual significa que hay una pulsación del reloj cada 0.000 000 286 segundos. La instrucción más rápida Z-80 tarda en ejecutarse cuatro pulsaciones de reloj, y la más lenta tarda unas 21 pulsaciones.

De lo anterior puedes deducir aproximadamente lo que tarda en ejecutarse una rutina sumando todos los tiempos de ejecución de las instrucciones que forman dicha rutina. Como dichos tiempos vienen dados en "ciclos", lo que tendrás que hacer después de sumar todos los tiempos será multiplicar el resultado por el valor anterior. Así obtendrás en segundos el tiempo de ejecución de una rutina.

Recuerda que las instrucciones que forman los bucles se ejecutan más de una vez.

Apéndice 3

Conversiones de hexadecimal a decimal y viceversa

Las tablas siguientes han sido diseñadas para ayudarte a pasar números de decimal a hexadecimal y viceversa. De todas maneras, recuerda que puedes usar las funciones &H y HEX$ en tu ordenador.

Tabla de conversión hexadecimal-decimal

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00XX	XX00
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	135	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	273	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

Para usar esta tabla correctamente, lo mejor es que leas el siguiente ejemplo: Vamos a buscar el equivalente en hexadecimal del número decimal 6200.

Lo primero que vamos a hacer es determinar el número binario de 16 bits. Así:

bbbbbbbb	bbbbbbbb
HOB	LOB

1. De la columna de la izquierda de la tabla bajo el epígrafe XX00 encontramos que el 6200 está entre 4096 y 8192; así que lo que hacemos es elegir el valor menor, es decir, el 4096, y del valor de la fila tomamos los cuatro bits más significativos del HOB, que quiere decir *High Order Byte,* byte de orden mayor, en español, para 1; en este caso, 0001

0001bbbb	bbbbbbbb
HOB	LOB

2. El segundo paso consiste en determinar los cuatro bits menos significativos del HOB; la diferencia entre 6200 y 4096 es 2140. Como la diferencia es todavía mayor que 255, vamos a la columna de la izquierda de la tabla encabezada por 00XX. Como ves, 2104 está entre 2048 y 2304; nuevamente vuelves a tomar el valor menor, es decir, el 2048, y vamos a la fila del valor y vemos que los cuatro bytes menos significativos del LOB es el 8, es decir, el 1000.

00011000	bbbbbbbb
HOB	LOB

3. En el siguiente paso tenemos que hallar el LOB, es decir, el *Low Order Byte*, que en español sería el byte de orden menor, del número.

Vemos que la diferencia entre 2104 y 2048 es 56. Dentro de la tabla vemos que el 56 está en la intersección de la fila 3 y la columna 8, por lo que tomamos como LOB el 38H:

00011000	00111000
HOB	LOB

El valor en hexadecimal del número decimal 6200 es 1838H.

La tabla siguiente da los equivalentes en complemento a dos de los números decimales.

Tabla de conversión decimal-hexadecimal en complemento a dos

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	−128	−127	−126	−125	−124	−123	−122	−121	−120	−119	−118	−117	−116	−115	−114	−113
9	−112	−111	−110	−109	−108	−107	−106	−105	−104	−103	−102	−101	−100	− 99	− 98	− 97
A	− 96	− 95	− 94	− 93	− 92	− 91	− 90	− 89	− 88	− 87	− 86	− 85	− 84	− 83	− 82	− 81
B	− 80	− 79	− 78	− 77	− 76	− 75	− 74	− 73	− 72	− 71	− 70	− 69	− 68	− 67	− 66	− 65
C	− 64	− 63	− 62	− 61	− 60	− 59	− 58	− 57	− 56	− 55	− 54	− 53	− 52	− 51	− 50	− 49
D	− 48	− 47	− 46	− 45	− 44	− 43	− 42	− 41	− 40	− 39	− 38	− 37	− 36	− 35	− 34	− 33
E	− 32	− 31	− 30	− 29	− 28	− 27	− 26	− 25	− 24	− 23	− 22	− 21	− 20	− 19	− 18	− 17
F	− 16	− 15	− 14	− 13	− 12	− 11	− 10	− 9	− 8	− 7	− 6	− 5	− 4	− 3	− 2	− 1

Apéndice 4

Instrucciones del Z-80

MNEMONICO	HEXADECIMAL	MNEMONICO	HEXADECIMAL	MNEMONICO	HEXADECIMAL
ADC A, (HL)	8E	ADD IY,IY	FD 29	BIT 2,A	CB 57
ADC A, (IX+dis)	DD 8E XX	ADD IY,SP	FD 39	BIT 2,B	CB 50
ADC A,(IY+dis)	FD 8E xx	AND (HL)	A6	BIT 2,C	CB 51
ADC A,A	8F	AND (IX+dis)	DD A6 XX	BIT 2,D	CB 52
ADC A,B	88	AND (IY+dis)	FD A6 XX	BIT 2,E	CB 53
ADC A,C	89	AND A	A7	BIT 2,H	CB 54
ADC A,D	8A	AND B	A0	BIT 2,L	CB 55
ADC A,n	CE XX	AND C	A1	BIT 3,(HL)	CB 5E
ADC A,E	8B	AND D	A2	BIT 3,(IX+dis)	DD CB XX 5E
ADC A,H	8C	AND n	E6 XX	BIT 3,(IY+dis)	FD CB XX 5E
ADC A,L	8D	AND E	A3	BIT 3,A	CB 5F
ADC HL,BC	ED 4A	AND H	A4	BIT 3,B	CB 58
ADC HL,DE	ED 5A	AND L	A5	BIT 3,C	CB 59
ADC HL,HL	ED 6A	BIT 0,(HL)	CB 46	BIT 3,D	CB 5A
ADC HL,SP	ED 7A	BIT 0,(IX+dis)	DD CB XX 46	BIT 3,E	CB 5B
ADD A, (HL)	86	BIT 0,(IY+dis)	FD CB XX 46	BIT 3,H	CB 5C
ADD A,(IX+dis)	DD 86XX	BIT 0,A	CB 47	BIT 3,L	CB 5D
ADD A,(IY+dis)	FD 86XX	BIT o,B	CB 40	BIT 4,(HL)	CB 66
ADD A,A	87	BIT 0,C	CB 41	BIT 4,(IX+dis)	DD CB XX 66
ADD A,B	80	BIT 0,D	CB 42	BIT 4,(IY+dis)	FD CB XX 66
ADD A,C	81	BIT 0,E	CB 43	BIT 4,A	CB 67
ADD A,D	82	BIT 0,H	CB 44	BIT 4,B	CB 60
ADD A,n	C6 XX	BIT 0,L	CB 45	BIT 4,C	CB 61
ADD A,E	83	BIT 1,(HL)	CB 4E	BIT 4,D	CB 62
ADD A,H.	84	BIT 1,(IX+dis)	DD CB XX 4E	BIT 4,E	CB 63
ADD A,L	85	BIT 1,(IY+dis)	FD CB XX 4E	BIT 4,H	CB 64
ADD HL,BC	09	BIT 1,A	CB 4F	BIT 4,L	CB 65
ADD HL,DE	19	BIT 1,B	CB 48	BIT 5,(HL)	CB 6E
ADD HL,HL	29	BIT 1,C	CB 49	BIT 5,(IX+dis)	DD CB XX 6E
ADD HL,SP	39	BIT 1,D	CB 4A	BIT 5,(IY+dis)	FD CB XX 6E
ADD IX,BC	DD 09	BIT 1,E	CB 4B	BIT 5,A	CB 6F
ADD IX,DE	DD 19	BIT 1,H	CB 4C	BIT 5,B	CB 68
ADD IX,IX	DD 29	BIT 1,L	CB 4D	BIT 5,C	CB 69
ADD IX,SP	DD 39	BIT 2,(HL)	CB 56	BIT 5,D	CB 6A
ADD IY,BC	FD 09	BIT 2,(IX+dis)	DD CB XX 56	BIT 5,E	CB 6B
ADD IY,DE	FD 19	BIT 2,(IY+dis)	FD CB XX 56	BIT 5,H	CB 6C

MNEMONICO	HEXADECIMAL
BIT 5,L	CB 6D
BIT 6,(HL)	CB 76
BIT 6,(IX+dis)	DD CB XX 76
BIT 6,(IY+dis)	FD CB XX 76
BIT 6,A	CB 77
BIT 6,B	CB 70
BIT 6,C	CB 71
BIT 6,D	CB 72
BIT 6,E	CB 73
BIT 3,H	CB 74
BIT 6,L	CB 75
BIT 7,(HL)	CB 7E
BIT 7,(IX+dis)	DD CB XX 7E
BIT 7,(IY+dis)	FD CB XX 7E
BIT 7,A	CB 7F
BIT 7,B	CB 78
BIT 7,C	CB 79
BIT 7,D	CB 7A
BIT 7,E	CB 7B
BIT 7,H	CB 7C
BIT 7,L	CB 7D
CALL ADDR	CD XX XX
CALL C,ADDR	DC XX XX
CALL M,ADDR	FC XX XX
CALL NC,ADDR	D4 XX XX
CALL NZ,ADDR	C4 XX XX
CALL P,ADDR	F4 XX XX
CALL PE,ADDR	EC XX XX
CALL PO,ADDR	E4 XX XX
CALL Z,ADDR	CC XX XX
CCF	3F
CP (HL)	BE
CP (IX+dis)	DD BE XX
CP (IY+dis)	FD BE XX
CP A	BF
CP B	B8
CP C	B9
CP D	BA
CP n	FE XX
CP E	BB
CP H	BC
CP L	BD
CPD	ED A9
CPDR	ED B9
CPI	ED A1
CPIR	ED B1
CPL	2F
DAA	27
DEC (HL)	35
DEC (IX+dis)	DD 35 XX
DEC (IY+dis)	FD 35 XX
DEC A	3D
DEC B	05
DEC BC	0B
DEC C	0D
DEC D	15
DEC DE	1B
DEC E	1D
DEC H	25
DEC HL	2B
DEC IX	DD 2B
DEC IY	FD 2B
DEC L	2D
DEC SP	3B
DI	F3
DJNZ,dis	10 XX
EI	FB
EX (SP),HL	E3
EX (SP),IX	DD E3
EX (SP),IY	FD E3
EX AF,AF'	08
EX DE,HL	EB
EXX	D9
HALT	76
IM 0	ED 46
IM 1	ED 56
IM 2	ED 5E
IN A,(C)	ED 78
IN A,port	DB XX
IN B,(C)	ED 40

MNEMONICO	HEXADECIMAL
IN C,(C)	ED 48
IN D,(C)	ED 50
IN E,(C)	ED 58
IN H,(C)	ED 60
IN L,(C)	ED 68
INC (HL)	34
INC (IX+dis)	DD 34 XX
INC (IY+dis)	FD 34 XX
INC A	3C
INC B	04
INC BC	03
INC C	0C
INC D	14
INC DE	13
INC E	1C
INC H	24
INC HL	23
INC IX	DD 23
INC IY	FD 23
INC L	2C
INC SP	33
IND	ED AA
INDR	ED BA
INI	ED A2
INIR	ED B2
JP (HL)	E9
JP (IX)	DD E9
JP (IY)	FD E9
JP ADDR	C3 XX XX
JP C,ADDR	DA XX XX
JP M,ADDR	FA XX XX
JP NC,ADDR	D2 XX XX
JP NZ,ADDR	C2 XX XX
JP P,ADDR	F2 XX XX
JP PE,ADDR	EA XX XX
JP PO,ADDR	E2 XX XX
JP Z,ADDR	CA XX XX
JR C,dis	38 XX
JR dis	18 XX
JR NC,dis	30 XX
JR NZ,dis	20 XX
JR Z,dis	28 XX
LD (ADDR),A	32 XX XX
LD(ADDR),BC	ED 43 XX XX
LD (ADDR),DE	ED 53 XX XX
LD(ADDR),HL	ED 63 XX XX
LD (ADDR),HL	22 XX XX
LD (ADDR),IX	DD 22 XX XX
LD (ADDR),IY	FD 22 XX XX
LD (ADDR),SP	ED 73 XX XX
LD (BC),A	02
LD (DE),A	12
LD (HL),A	77
LD (HL),B	70
LD (HL),C	71
LD (HL),D	72
LD (HL),n	36 XX
LD (HL),E	73
LD (HL),H	74
LD (HL),L	75
LD (IX+dis),A	DD 77 XX
LD (IX+dis),B	DD 70 XX
LD (IX+dis),C	DD 71 XX
LD (IX+dis),D	DD 72 XX
LD (IX+dis),n	DD 36 XX XX
LD (IX+dis),E	DD 73 XX
LD (IX+dis),H	DD 74 XX
LD (IX+dis),L	DD 75 XX
LD (IY+dis),A	FD 77 XX
LD (IY+dis),B	FD 70 XX
LD (IY+dis),C	FD 71 XX
LD (IY+dis),D	FD 72 XX
LD (IY+dis),n	FD 36 XX XX
LD (IY+dis),E	FD 73 XX
LD (IY+dis),H	FD 74 XX
LD (IY+dis),L	FD 75 XX
LD A,(ADDR)	3A XX XX
LD A,(BC)	0A
LD A,(DE)	1A
LD A,(HL)	7E
LD A,(IX+dis)	DD 7E XX

MNEMONICO	HEXADECIMAL
LD A,(IY+dis)	FD 7E XX
LD A,A	7F
LD A,B	78
LD A,C	79
LD A,D	7A
LD A,n	3E XX
LD A,E	7B
LD A,H	7C
LD A,I	ED 57
LD A,L	7D
LD A,R	ED 5F
LD B,(HL)	46
LD B,(IX+dis)	DD 46 XX
LD B,(IY+dis)	FD 46 XX
LD B,A	47
LD B,B	40
LD B,C	41
LD B,D	42
LD B,n	06 XX
LD B,E	43
LD B,H	44
LD B,L	45
LD BC,(ADDR)	ED 4B XX XX
LD BC,nn	01 XX XX
LD C,(HL)	4E
LD C,(IX+dis)	DD 4E xx
LD C,(IY+dis)	FD 4E XX
LD C,A	4F
LD C,B	48
LD C,C	49
LD C,D	4A
LD C,n	0E XX
LD C,E	4B
LD C,H	4C
LD C,L	4D
LD D,(HL)	56
LD D,(IX+dis)	DD 56 XX
LD D,(IY+dis)	FD 56 XX
LD D,A	57
LD D,B	50
LD D,C	51
LD D,D	52
LD D,n	16 XX
LD D,E	53
LD D,H	54
LD D,L	55
LD DE,(ADDR)	ED 5B XX XX
LD DE,nn	11 XX XX
LD E,(HL)	5E
LD E,(IX+dis)	DD 5E XX
LD E,(IY+dis)	FD 5E XX
LD E,A	5F
LD E,B	58
LD E,C	59
LD E,D	5A
LD E,n	1E XX
LD E,E	5B
LD E,H	5C
LD E,L	5D
LD H,(HL)	66
LD H,(IX+dis)	DD 66 XX
LD H,(IY+dis)	FD 66 XX
LD H,A	67
LD H,B	60
LD H,C	61
LD H,D	62
LD H,n	26 XX
LD H,E	63
LD H,H	64
LD H,L	65
LD HL,(ADDR)	ED 6B XX XX
LD HL,(ADDR)	2A XX XX
LD HL,nn	21 XX XX
LD I,A	ED 47
LD IX,(ADDR)	DD 2A XX XX
LD IX,nn	DD 21 XX XX
LD IY (ADDR)	FD 2A XX XX
LD IY,nn	FD 21 XX XX
LD L,A	6F
LD L,B	68

MNEMONICO	HEXADECIMAL	MNEMONICO	HEXADECIMAL	MNEMONICO	HEXADECIMAL
LD L,C	69	RES 2,B	CB 90	RLC (IY+dis)	FD CB XX 06
LD L,D	6A	RES 2,C	CB 91	RLC A	CB 07
LD L,n	2E XX	RES 2,D	CB 92	RLC B	CB 00
LD L,E	6B	RES 2,E	CB 93	RLC C	CB 01
LD L, (HL)	6E	RES 2,H	CB 94	RLC D	CB 02
LD L,(IX+dis)	DD 6E XX	RES 2,L	CB 95	RLC E	CB 03
LD L, (IY+dis)	FD 6E XX	RES 3, (HL)	CB 9E	RLC H	CB 04
LD L,H	6C	RES 3, (IX+dis)	DD CB XX 9E	RLC L	CB 05
LD L,L	6D	RES 3, (IY+dis)	FD CB XX 9E	RLCA	07
LD R,A	ED 4F	RES 3,A	CB 9F	RLD	ED 6F
LD SP, (ADDR)	ED 7B XX XX	RES 3,B	CB 98	RR (HL)	CB 1E
LD SP,nn	31 XX XX	RES 3,C	CB 99	RR (IX+dis)	DD CB XX 1E
LD SP,HL	F9	RES 3,D	CB 9A	RR (IY+dis)	FD CB XX 1E
LD SP,IX	DD F9	RES 3,E	CB 9B	RR A	CB 1F
LD SP,IY	FD F9	RES 3,H	CB 9C	RR B	CB 18
LDD	ED A8	RES 3,L	CB 9D	RR C	CB 19
LDDR	ED B8	RES 4, (HL)	CB A6	RR D	CB 1A
LDI	ED A0	RES 4, (IX+dis)	DD CB XX A6	RR E	CB 1B
LDIR	ED B0	RES 4, (IY+dis)	FD CB XX A6	RR H	CB 1C
NEG	ED 44	RES 4,A	CB A7	RR L	CB 1D
NOP	00	RES 4,B	CB A0	RRA	1F
OR (HL)	B6	RES 4,C	CB A1	RRC (HL)	CB 0E
OR (IX+dis)	DD B6 XX	RES 4,D	CB A2	RRC (IX+dis)	DD CB XX 0E
OR (IY+dis)	FD B6 xx	RES 4,E	CB A3	RRC (IY+dis)	FD CB XX 0E
OR A	B7	RES 4,H	CB A4	RRC A	CB 0F
OR B	B0	RES 4,L	CB A5	RRC B	CB 08
OR C	B1	RES 5 (HL)	CB AE	RRC C	CB 09
OR D	B2	RES 5, (IX+dis)	DD CB XX AE	RRC D	CB 0A
OR n	F6 XX	RES 5, (IY+dis)	FD CB XX AE	RRC E	CH 0B
OR E	B3	RES 5,A	CB AF	RRC H	CB 0C
OR H	B4	RES 5,B	CB A8	RRC L	CB 0D
OR L	B5	RES 5,C	CB A9	RRCA	0F
OTDR	ED BB	RES 5,D	CB AA	RRD	ED 67
OTIR	ED B3	RES 5,E	CB AB	RST 00	C7
OUT (C) ,A	ED 79	RES 5,H	CB AC	RST 08	CF
OUT (C) ,B	ED 41	RES 5,L	CB AD	RST 10	D7
OUT (C) ,C	ED 49	RES 6, (HL)	CB B6	RST 18	DF
OUT (C) ,D	ED 51	RES 6, (IX+dis)	DD CB XX B6	RST 20	E7
OUT (C) ,E	ED 59	RES 6, (IY+dis)	FD CB XX B6	RST 28	EF
OUT (C) ,H	ED 61	RES 6,A	CB B7	RST 30	F7
OUT (C) ,L	ED 69	RES 6,B	CB B0	RST 38	FF
OUT part,A	D3 port	RES 6,C	CB B1	SBC A, (HL)	9E
OUTD	ED AB	RES 6,D	CB B2	SBC A, (IX+dis)	DD 9E XX
OUTI	ED A3	RES 6,E	CB B3	SBC A, (IY+dis)	FD 9E XX
POP AF	F1	RES 6,H	CB B4	SBC A,A	9F
POP BC	C1	RES 6,L	CB B5	SBC A,B	98
POP DE	D1	RES 7, (HL)	CB BE	SBC A,C	99
POP HL	E1	RES 7, (IX+dis)	DD CB XX BE	SBC A,D	9A
POP IX	DD E1	RES 7, (IY+dis)	FD CB XX BE	SBC A,n	DE XX
POP IY	FD E1	RES 7,A	CB BF	SBC A,E	9B
PUSH AF	F5	RES 7,B	CB B8	SBC A,H	9C
PUSH BC	C5	RES 7,C	CB B9	SBC A,L	9D
PUSH DE	D5	RES 7,D	CB BA	SBC HL,BC	ED 42
PUSH HL	E5	RES 7,E	CB BB	SBC HL,DE	ED 52
PUSH IX	DD E5	RES 7,H	CB BC	SBC HL,HL	ED 62
PUSH IY	FD E5	RES 7,L	CB BD	SBC HL,SP	ED 72
RES 0, (HL)	CB 86	RET	C9	SCF	37
RES 0, (IX+dis)	DD CB XX 86	RET C	D8	SET 0, (HL)	CB C6
RES 0, (IY+dis)	FD CB XX 86	RET M	F8	SET 0, (IX+dis)	DD CB XX C6
RES 0,A	CB 87	RET NC	D0	SET 0, (IY+dis)	FD CB XX C6
RES 0,B	CB 80	RET NZ	C0	SET 0,A	CB C7
RES 0,C	CB 81	RET P	F0	SET 0,B	CB C0
RES 0,D	CB 82	RET PE	E8	SET 0,C	CB C1
RES 0,E	CB 83	RET PO	E0	SET 0,D	CB C2
RES 0,H	CB 84	RET Z	C8	SET 0,E	CB C3
RES 0,L	CB 85	RETI	ED 4D	SET 0,H	CB C4
RES 1, (HL)	CB 8E	RETN	ED 45	SET 0,L	CB C5
RES 1, (IX+dis)	DD CB XX 8E	RL (HL)	CB 16	SET 1, (HL)	CB CE
RES 1, (IY+dis)	FD CB XX 8E	RL (IX+dis)	DD CB XX 16	SET 1, (IX+dis)	DD CB XX CE
RES 1,A	CB 8F	RL (IY+dis)	FD CB XX 16	SET 1, (IY+dis)	FD CB XX CE
RES 1,B	CB 88	RL A	CB 17	SET 1,A	CB CF
RES 1,C	CB 89	RL B	CB 10	SET 1,B	CB C8
RES 1,D	CB 8A	RL C	CB 11	SET 1,C	CB C9
RES 1,E	CB 8B	RL D	CB 12	SET 1,D	CB CA
RES 1,H	CB 8C	RL E	CB 13	SET 1,E	CB CB
RES 1,L	CB 8D	RL H	CB 14	SET 1,H	CB CC
RES 2, (HL)	CB 96	RL L	CB 15	SET 1,L	CB CD
RES 2, (IX+dis)	DD CB XX 96	RLA	17	SET 2, (HL)	CB D6
RES 2, (IY+dis)	FD CB XX 96	RLC (HL)	CB 06	SET 2, (IX+dis)	DD CB XX D6
RES 2,A	CB 97	RLC (IX+dis)	DD CB XX 06	SET 2, (IY+dis)	FD CB XX D6

MNEMONICO	HEXADECIMAL	MNEMONICO	HEXADECIMAL	MNEMONICO	HEXADECIMAL
SET 2,A	CB D7	SET 6, (HL)	CB F6	SRA E	CB 2B
SET 2,B	CB D0	SET 6, (IX+dis)	DD CB XX F6	SRA H	CB 2C
SET 2,C	CB D1	SET 6, (IY+dis)	FD CB XX F6	SRA L	CB 2D
SET 2,D	CB D2	SET 6,A	CB F7	SRL (HL)	CB 3E
SET 2,E	CB D3	SET 6,B	CB F0	SRL (IX+dis)	DD CB XX 3E
SET 2,H	CB D4	SET 6,C	CB F1	SRL (IY+dis)	FD CB XX 3E
SET 2,L	CB D5	SET 6,D	CB F2	SRL A	CB 3F
SET 3, (HL)	CB DE	SET 6,E	CB F3	SRL B	CB 38
SET 3, (IX+dis)	DD CB XX DE	SET 6,H	CB F4	SRL C	CB 39
SET 3, (IY+dis)	FD CB XX DE	SET 6,L	CB F5	SRL D	CB 3A
SET 3,A	CB DF	SET 7, (HL)	CB FE	SRL E	CB 3B
SET 3,B	CB D8	SET 7, (IX+dis)	DD CB XX FE	SRL H	CB 3C
SET 3,C	CB D9	SET 7,(IY+dis)	FD CB XX FE	SRL L	CB 3D
SET 3,D	CB DA	SET 7,A	CB FF	SUB (HL)	96
SET 3,E	CB DB	SET 7,B	CB F8	SUB (IX+dis)	DD 96 XX
SET 3,H	CB DC	SET 7,C	CB F9	SUB (IY+dis)	FD 96 XX
SET 3,L	CB DD	SET 7,D	CB FA	SUB A	97
SET 4, (HL)	CBE6	SET 7,E	CB FB	SUB B	90
SET 4, (IX+dis)	DD CB XX E6	SET 7,H	CB FC	SUB C	91
SET 4, (IY+dis)	FD CB XX E6	SET 7,L	CB FD	SUB D	92
SET 4,A	CB E7	SLA (HL)	CB 26	SUB E	93
SET 4,B	CB E0	SLA (IX+dis)	DD CB XX 26	SUB n	D6 XX
SET 4,C	CB E1	SLA (IY+dis)	FD CB XX 26	SUB H	94
SET 4,D	CB E2	SLA A	CB 27	SUB L	95
SET 4,E	CB E3	SLA B	CB 20	XOR (HL)	AE
SET 4,H	CB E4	SLA C	CB 21	XOR (IX+dis)	DD AE XX
SET 4,L	CB E5	SLA D	CB 22	XOR (IY+dis)	FD AE XX
SET 5, (HL)	CB EE	SLA E	CB 23	XOR A	AF
SET 5, (IX+dis)	DD CB XX EE	SLA H	CB 24	XOR B	A8
SET 5, (IY+dis)	FD CB XX EE	SLA L	CB 25	XOR C	A9
SET 5,A	CB EF	SRA (HL)	CB 2E	XOR D	AA
SET 5,B	CB E8	SRA (IX+dis)	DD CB XX 2E	XOR n	EE XX
SET 5,C	CB E9	SRA (IY+dis)	FD CB XX 2E	XOR E	AB
SET 5,D	CB EA	SRA A	CB 2F	XSOR H	AC
SET 5,E	CB EB	SRA B	CB 28	XOR L	AD
SET 5,H	CB EC	SRA C	CB 29		
SET 5,L	CB ED	SRA D	CB 2A		

Apéndice 5

Símbolos usados

Símbolo	*Operación*
C	*Flag* de acarreo. C = 1 si la operación genera acarreo del bit más significativo del resultado o del operando.
Z	*Flag* 0. Z = 1 si el resultado de una operación es 0.
S	*Flag* de signo. S = 1 si el bit más significativo del resultado es 1, es decir, es un número negativo.
P/V	*Flag* de paridad/desbordamiento. La paridad (P) y el desbordamiento (V) comparten el mismo *flag*. Las operaciones lógicas afectan este *flag* con la paridad del resultado, mientras que las operaciones aritméticas afectan el *flag* con el desbordamiento del resultado. Si el *flag* P/V tiene paridad, P/V = 1 si el resultado de la operación es par, P/V = 0 si el resultado es impar. Si el *flag* P/V tiene el desbordamiento, P/V = 1 si el resultado de la operación genera desbordamiento.
H	*Flag* de medio acarreo. H = 1 si la operación de adición o substracción produce un acarreo del bit 4 del acumulador.
N	*Flag* de adición/substracción. N = 1 si la operación anterior fue una resta.
	Los *flags* H y N se usan junto con la instrucción de ajuste decimal DAA para devolver el resultado en formato empaquetado BCD cuando se usan operandos de suma o resta con formato empaquetado BCD.
	El *flag* queda afectado de acuerdo con el resultado de una operación.
*	La operación no ha afectado el *flag*.

Símbolo	*Operación*
0	La operación desactiva, pone a 0, el *flag*.
1	La operación activa, pone a 1, el *flag*.
?	El resultado del *flag* no se conoce.
V	El *flag* P/V es afectado de acuerdo con el desbordamiento de la operación.
P	El *flag* P/V es afectado de acuerdo con la paridad de la operación.
r	Cualquiera de los registros de la UCP siguientes: A, B, C, D, E, H, L.
s	Cualquier posición de 8 bits para todos los modos de direccionamientos situados en las instrucciones.
SS	Cualquier posición de 16 bits para todos los modos de direccionamientos situados en esa instrucción.
R	Registro de refresco.
n	Valor de 8 bits dentro del rango 0 a 255.
nn	Valor de 16 bits dentro del rango 0 a 65535.

Apéndice 6

Sumario de los *flags*

Tabla resumen de *flags*

Instrucción	C	Z	P/V	S	N	H	*Comentarios*
ADC HL,SS	*	*	V	*	0	X	Suma de 16 bits con acarreo
ADX s;ADD s	*	*	V	*	0	*	Suma de 8 bits o suma con acarreo
ADD DD,SS	*	—	—	—	0	X	Suma de 16 bits
AND s	0	*	P	*	0	1	Operaciones lógicas
BIT b,s	—	*	X	X	0	X	Estado del bit *b* en la posición *s*. Se copia en el *flag* 3
CCF	*	—	—	—	0	X	Complemento del acarreo
CPD;CPDR;CPI;CPIR;	—	*	*	X	1	X	Instrucción de búsqueda en bloques: Z=1 si A=(HL) si no Z=0 P/V=1 si BC≠0. En otro caso, P/V = 0
CP s	*	*	V	*	1	*	Compara el acumulador
CPL	—	—	—	—	1	1	Complementa el acumulador
DAA	*	*	P	*	—	*	Ajuste decimal del acumulador
							Decremento de 8 bits
IN r,(C)	—	*	P	*	0	0	Entrada indirecta de registros

Instrucción	C	Z	P/V	S	N	H	*Comentarios*
INC s	—	*	V	*	0	*	Incremento con 8 bits
IND;INI	—	*	X	X	1	X	Entrada de bloques Z=0 si B=0 si no Z=1
INDR:INIR	—	1	X	X	1	X	Entrada de bloques Z=0 si B≠0 si no Z=1
LD A,I;LD A,R	—	*	IFF	*	0	0	Contenido de *switch* de interrupciones activas. Se copia en el registro P/V
LDD;LDI	—	X	*	X	0	0	Instrucciones de transferencia de bloques
LDDR;LDIR	—	X	0	X	0	0	P/V=1 si BC≠0, si no P/V=0
NEG	*	*	V	*	1	*	Negación del acumulador
OR s	0	*	P	*	0	0	"O" lógico del acumulador
OTDR;OTIR	—	*	X	X	1	X	Salida de bloques Z=0 si B≠0. En otro caso, Z=1
OUTD;OUTI	—	*	X	X	1	X	Salida de bloques Z=0 si B≠0. En otro caso, Z=1
RLA;RLCA;RRA;RRCA	*	—	—	—	0	0	Rotación del acumulador
RLD;RRD	—	*	P	*	0	/	Rotación derecha e izquierda
RLS;RLCs;RRs;RRCs SLAs;SRAs;SRLs	*	*	P	*	0	0	Rotación y desplazamiento
SBC HL,SS	*	*	V	*	1	X	Resta de 16 bits con acarreo
SCF	1	—	—	—	0	0	Activa el *flag* de acarreo
SBCs;SUD s			V		1		Resta con 8 bits con acarreo
XOR x	0		P		0	0	"O" exclusivo con el acumulador

Indice alfabético

Los *chips* de tu MSX: CPU Z80/PSG AY-3-8910/VDP TMS9918 tienen unas tremendas posibilidades que sólo podrás explotar si programas en lenguaje máquina.

¡Escapa de los confines del BASIC! Sólo trabajando en código máquina aprovecharás la operatividad, velocidad y potencia reales de tu micro MSX.

LENGUAJE MAQUINA PARA MSX te enseñará qué es el acumulador, los registros índices o el puntero de pila, qué es y cómo se recupera información de una pila o *stack,* cómo se manejan y para qué sirven las interrupciones.

Pero no sólo aprenderás cómo funciona y cuál es la estructura del Z80; también conocerás a fondo la estructura interna del ordenador: su mapa de memoria, puertos de entrada/salida y cómo programar el VDP (manejo de pantalla, gráficos, *sprites*) o el PSG (generador programable de sonido).

LENGUAJE MAQUINA PARA MSX te da una panorámica profunda de la programación de tu MSX que te permitirá aprender rápidamente código máquina y desarrollar aplicaciones sofisticadas.

¡Adelante, los *chips* te esperan!

